Implementing compression on distributed time series database

Michael Burman

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 05.11.2017

Supervisor

Prof. Kari Smolander

Advisor

Mgr. Jiri Kremser

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Michael Burman

Title Implementing compression on distributed time series database

Degree programme

Major Computer Science **Code of major** SCI3042

Supervisor Prof. Kari Smolander

Advisor Mgr. Jiri Kremser

Date 05.11.2017 **Number of pages** 70+4 **Language** English

Abstract

Rise of microservices and distributed applications in containerized deployments are putting increasing amount of burden to the monitoring systems. They push the storage requirements to provide suitable performance for large queries.

In this paper we present the changes we made to our distributed time series database, Hawkular-Metrics, and how it stores data more effectively in the Cassandra. We show that using our methods provides significant space savings ranging from 50 to 95% reduction in storage usage, while reducing the query times by over 90% compared to the nominal approach when using Cassandra.

We also provide our unique algorithm modified from Gorilla compression algorithm that we use in our solution, which provides almost three times the throughput in compression with equal compression ratio.

Keywords timeseries compression performance storage

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Michael Burman

Työn nimi Pakkausmenetelmät hajautetussa aikasarjatietokannassa

Koulutusohjelma

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja ja ohjaaja Prof. Kari Smolander

Päivämäärä 05.11.2017 **Sivumäärä** 70+4 **Kieli** Englanti

Tiivistelmä
Hajautettujen järjestelmien yleistyminen on aiheuttanut valvontajärjestelmissä tiedon määrän kasvua, sillä aikasarjojen määrä on kasvanut ja niihin talletetaan useammin tietoa. Tämä on aiheuttanut kasvavaa kuormitusta levyjärjestelmille, joilla on ongelmia palvella kasvavia kyselyitä.

Tässä paperissa esittelemme muutoksia hajautettuun aikasarjatietokantaamme, Hawkular-Metricsiin, käyttäen hyödyksi tehokkaampaa tiedon pakkausta ja järjestelyä kun tietoa talletetaan Cassandraan. Nopeutimme kyselyjä lähes kymmenkertaisesti ja samalla pienensimme levytilavaatimuksia aineistosta riippuen 50-95%.

Esittelemme myös muutoksemme Gorilla pakkausalgoritmiin, jota hyödynnämme tulosten saavuttamiseksi. Muutoksemme nopeuttavat pakkaamista melkein kolminkertaiseksi alkuperäiseen algoritmiin nähden ilman pakkaustehon laskua.

Avainsanat aikasarjat suorituskyky pakkausmenetelmät

Preface

I would like to thank Red Hat for providing me with an interesting problem to work on and my team for providing valuable feedback. Special thanks go to John Sanda for great discussions, code reviews and ideas and to Filip Brychta for relentless quality testing. I also like to thank Kari Smolander for feedback on the thesis.

In the end, this paper would not have been possible without the support from my wife and my kids, Linnea, Elli and Nella.

Otaniemi, 5.11.2017

Michael Burman

Contents

Symbols and abbreviations

Symbols

δ delta

Operators

$\oplus$ eXclusive OR

Abbreviations

GC	Garbage Collector
JIT	Just In Time compiler
JVM	Java Virtual Machine
ODS	Facebook's Operational Data Store
LSM	Log-Structured Merge Tree
TSM	Time-Structured Merge Tree
RLE	Run-Length Encoding
FFT	Fast Fourier Transform
LZ77	Lempel and Ziv compression algorithm from the 1977 paper
LZ78	Lempel and Ziv compression algorithm from the 1978 paper
FOR	Frame-Of-Reference
PFOR	Patched Frame-Of-Reference
AFOR-1	Delbru et al. 32-integer version of binary packing
AFOR-3	Greedy algorithm for binary packing partitioning
VSEncoding	Vector of Split Encoding
FCM	Finite Context Method Predictors
DFCM	Differential FCM
ANS	Asymmetric Numeral System
FSE	Finite State Entropy
Zstd	Zstandard
NSM	n-ary storage model
DSM	decomposite storage model
SLP	Superword Level Parallelism

1 Introduction

Distributed applications are generating increasing amount of data given all the modern possibilities of instrumenting an application. This data must be stored and processed for queries that require historical knowledge, such as autoscaling based on the historical usage data or when it is used by the customers for internal or external billing purposes, in which case the exact data must be kept and stored with multiple copies.

Containers, which are isolated application images that package everything needed to run an application, including code and runtime requirements[126] have been the hype in the infrastructure development for few years now. To manage large amount of containers, solutions such as Kubernetes[127], which is a solution for automating deployment, management and scaling of containers, have risen in popularity. Due to the lower overhead of containers compared to the virtual machines, it has allowed infrastructure to run workloads with higher efficiency. Containerized systems have been a natural deployment system for microservices and lambda architectures, thus driving up the amount of components in the system.

These changes have required the monitoring systems to digest more and more data all the time, quickly growing the amount of ingested data points and unique time series. Traditional time series storage solutions such as Graphite[48] or relational databases were not designed for these rates of data and are showing their limitations in terms of storage efficiency and query processing speeds. These changes in the industry have given raised interest to develop[65][92][96][32][105][97] efficient compression methods to operational time series storage. Compression is an effective[77][92][136][97] way of reducing the required data footprint to reduce memory requirements and I/O pressure.

The focus of this paper is to detail our changes our changes to the Hawkular-Metrics, a stateless distributed time series database that uses Cassandra as the underlying storage engine, to allow reducing the amount of data that is stored on the disk and how to query that data. While Cassandra provides a generic block compression, this is not effective way of compressing monitoring data. We show how our custom built processes take a stream of data and modifies it from a Cassandra's row model to a highly efficient columnar model. We follow the model of a hybrid columnar database which uses a combination of row modeling to quickly store the potentially volatile data and then transforms it to a columnar layout in the long term storage. Due to the columnar layout, we are able to better compress the data since data points are adjacent to each other.

In short, we want our solution to fullfill the following requirements:

1. Find the most effective way of compressing the operational time series without noticeable write performance hit

2. Store the data to Cassandra without complicating operational behavior, while keeping all the advantages of the distributed storage without a single point of failure.

3. Implement the solution to Hawkular-Metrics, while retaining its stateless architecture.

4. All the state must be stored in the Cassandra

The solution provided here is not a general solution for all the distributed databases when storing time series data. The architecture can be applied to all the solutions that are built like Cassandra with the constraints from the Dynamo[30], such as ScyllaDB[95] or Dynomite[86] and where the data is stored in a Log-Structured Merge-Tree[87]. The compression results should be applicable on the other hand to all the operational time series, but may not be necessarily the optimal solution for all types of time series data.

We also show the compression efficiency of multiple compression formats, ranging from generic compression algorithms to specific ones that are optimized for floating point data or integer data. We present our changes to the Gorilla time series compression algorithm[92] and how we improved the existing algorithm's performance. We provide all our work in open-source with a friendly license.

1.1 Outline

This thesis consists of two background sections and two sections for contributions. Section 2 discusses the backgrounds of the problems this thesis is trying to solve while the Section 3 discusses the different compression techniques. Section 4 is for evaluation methods and criterias, while Section 5 is dedicated to the description of the solution and Section 6 talks about the results that were achieved. Section 7 is dedicated to conclusions. Appendix A describes the optimizations techniques for Java and Appendix B provides the necessary links to access the source code of all contributions.

2 Preliminaries

2.1 Time series

Data sets which are collections of measurements over time are called time series. Time series are multi-dimensional data, which have a sequence of numbers $x_1, x_2, .., x_n$ where each value has an attribute at successive points in time. [100] [97] Time series data can be real or discrete valued and can have multiple additional dimensions describing it.

Time series are used in many applications, such as financial data, sensor data, health data, monitoring systems and scientific measurements.[97] In this paper, we concentrate mainly on the operational time series which are generated by the monitoring systems. The area of storing monitoring data has been a hot topic lately in the research [76][105][92][35][32] and there are also a lot of popular projects around it such as [65][91][96][84][132]. The monitoring data has number of distinct characteristics which are exploited by the specialized tools:

- Individual values are of very little interest compared to aggregations [76][92][105]
- Data points are constantly spaced in the time series[92][76][35]
- Frequency of the time spans varies between time series[35][76]
- Abrupt fluctuations and randomicity are common [35][32]
- Huge variance of data between metrics [35]
- Data is immutable[125][113]
- Data can be lost[125][113]

2.1.1 Time series storage

A time series storage is a database optimized for the storage and querying of time series data. Time series data can be stored in a specialized container inside underlying storage systems, including relational[122][62][59] and non-relational databases[76][91][70], but also with custom storage layers[92][65][96][107][48].

Time series data does not fit the generic purpose storage solutions well as it has its own distinct characteristics. The generality of relational databases limits the ingestion speeds[125][32] as well as provides features that are not required when processing the time series[113][76].

The stored time series are often associated with additional metadata that describes the time series[97]. The metadata can be used in the querying to search or group similar series. In many systems these are called tags[91][50] or labels[96]. Older systems[48] used a special notation, such as dot notation, in their metric naming to provide similar metadata. Some systems are planning to take these even further by removing the idea of metric name and instead giving each metric a graph of metadata[83].

Three main components of the time series database are[97] data ingestion layer, query processor and storage engine. Data is stored to these systems often by the means of collectors, such as [52][116][55], which operate on multiple machines and feed the data continuously to the system. These collectors can be simple and with limited processing capabilities and they will push the data in raw format to the storage.

There is no common query language for the time series and each time series database has their own query language. Examples include functional access[48][96], languages like SQL[65] and some are using REST-like interfaces[91][50]. Very common queries include, but are not limited to[97][113]:

- Query using metadata information and time
- Transforming the data by aggregation
- Visualization of the data

Data is usually[92][97] aggregated before processing it. It is also possible to pre-aggregate results to improve read performance[91].

2.2 Hawkular

Hawkular[50] is a Red Hat sponsored open source project targeted for monitoring solutions. It consists of multiple components to provide solutions for monitoring requirements and can be customized for different projects depending on their needs. For example, ManageIQ[82] uses full Hawkular Services solution in the middleware management component, while OpenShift[90] uses Hawkular Metrics component only. The third main distribution model of Hawkular is Hawkular APM, which provides application management capabilities by tracing requests.

2.2.1 Hawkular-Metrics

Hawkular-Metrics is a time series storage that underneath stores all the data to the Cassandra[5]. While Cassandra itself is suitable for storing the data, it lacks features that make it easy to query for data and has limited data modeling properties. As such, Hawkular-Metrics is intended to be a gateway to storing data by taking advatange of the Cassandra's distributed nature, but helping the user to avoid potential pitfalls and to reimplement all the querying features.

Hawkular-Metrics is built using stateless architecture[51], taking advantage of asynchronous streaming processing with backpressure support. Each time series is considered a continuous stream of data points with unbounded growth possibility. The implementation is built by using the RxJava[103], a ReactiveX[102] implementation for Java. This architecture allows flexible deployment models, where the user can have any combination of Hawkular-Metrics instances or Cassandra instances, depending on the bottleneck or requirements for processing. This provides a highly available and a scalable solution for time series processing. The data is ingested through a REST-interface with JSON encoding.

Hawkular-Metrics provides a rich set of features for managing the time series. Multi-tenancy is one of the core features and each request requires a tenant to be provided. This tenancy is used to build the data model also, thus allowing separation of data for each tenant, and is actively used in Openshift[39] for security purposes, as each user has a limited set of projects they can access, which we then map to a tenant. Thus, the user is unable to access data that is not part of any project he is participating. Authorization is checked on each request from the Openshift to ensure that the access token still allows accessing the data. Each of the time series is defined by a set of tags that allows grouping, sorting and finding time series. The data is automatically removed after a per time series configurable retention time.

2.3 Cassandra

Cassandra is a distributed database that is designed to be highly available without a single point of failure spread across hundreds of nodes and to store large amounts of data. One of the key designs is the ability to withstand hardware failures and network partitions, with the performance characteristics to scale linearly. Scaling requires efficient data partitioning strategies. [74] Cassandra is built by using a peer-to-peer network that shares the data among all the nodes in the cluster and the state information is shared between the nodes using a peer-to-peer gossip protocol. Reads and writes can be targeted to any node in the cluster, and that node then becomes a coordinator and a proxy for those operations.

All the writes are partitioned and replicated around the cluster. [9] The data is stored on multiple replicas to ensure fault tolerance and reliability with a replication strategy that determines which nodes store the replicas and how many replicas are stored, in which datacenters. All replicas are considered equal, there are no master or slaves for the data [28]. To ensure data integrity a repair job is periodically ran, which checks the consistency of the data in the cluster. [9]

A partitioner determines how the data is distributed inside the cluster. It calculates a hash from the partition key of the stored row and uses the hash as a token that indicates the placement of the data in the cluster. Each node has a token range they are responsible for and consistent hashing is used to minimize the amount of data movement required when new nodes join or leave. [25]

2.3.1 Storage

Cassandra is using a storage that is built to resemble a Log-Structured Merge-tree (LSM)[87] and to avoid reading data before writing it[118]. LSM trees provide performance characteristics that are suitable for high insert volumes while still maintaining indexable structure. To avoid doing a read-write, each Cassandra write is first to written to a Write-Ahead Log (WAL) on the disk and to an in-memory structure that is called a memtable, which buffers the writes. The memtable stores writes in a sorted order in the memory until reaching a configurable limit, at which point the memtables are written to the disk as SSTables, which are immutable.[58]

Deletes are handled like updates and inserts to support the Cassandra's data

distribution processes. The data that is being deleted is written to the data store as a tombstone record. A tombstone has a time-to-live (TTL) that is different from a normal cell TTL. When normal row's TTL expires, a tombstone is written for that row, however when tombstone reaches its TTL, a compaction process is allowed to remove the row from physical storage. [56]

The data is stored by serializing the internal structure, which is divided among partitions and rows. A partition is a collection of rows with the same partition key and inside the partition the rows are sorted by their clustering key. Each row is then identified by their primary key. [67] Updates to a row, tombstones or normal updates can cause different versions of row being written to multiple SSTables in the disk. To retrieve a row from the disk, Cassandra must read each version and to do that it must be read multiple SSTables.

To reduce the amount of SSTables that must be read to fetch a row and to remove the duplicate versions, Cassandra employs a process called compaction. Compaction works on multiple SSTables by merging multiple versions of a row to a single version based on write timestamp and writing it to a new fresh SSTable. While the process does not use random I/O because of the partition key ordering, it does cause I/O spikes in the system. As SSTables are immutable, old SSTables are deleted after they have been compacted and all read requests from them are completed. Cassandra supports multiple strategies on how to select which SSTables are selected for compaction and how the resulting SSTables are created.[57]

In 2014, Cassandra got a new compaction strategy called Date Tiered Compaction Strategy (DTCS) for time series data. While the idea was to reduce write amplification (a process where data needs to be written more than once) there were several implementation details that made it difficult to use in the real world.[124] A replacement was committed to Cassandra in 2016[20] called Time Windows Compaction Strategy (TWCS) which also caused the deprecation of the DTCS.

TWCS is an extension of the Size Tiered Compaction Strategy (STCS)[57] by creating a buckets of time that are compacted using the STCS while the window is open. STCS works by compacting similar sized SSTables into larger SSTables. When the time window closes, the TWCS will compact all the SSTables in the window to a single SSTable, which is then never compacted again. [124]

This process gives it some limitations that makes it unsuitable for scenarios where deletes or updates happen to the rows after the time window has been closed as those rows are then never compacted. This limits it to the use cases where data is written once and expired using a TTL.

2.4 Related work

Due to the increased interest in the time series data in the industry, the open source communities have created a large amount of time series databases to handle different workloads. To our knowledge, none of the distributed open source databases however provide persistent compressed storage for long-term retention. In terms of features, the user would have to choose between the persistence, distributed nature or compression capabilities without any solution providing all three.

2.4.1 Gorilla / Beringei

Facebook's Gorilla paper[92] describing their in-memory time series database was based on their insights, observations and requirements while monitoring the Facebook infrastructure. To improve their query and resource efficiency, they developed compression methods which they evaluated against the data stored in their current ODS (Operational Data Store) solution to optimize the compression scheme for their internal usage.

In their architecture, the data is stored with 64 bit timestamp and 64 bit floating point value while the data can be located with a simple string key. This same structure forms the basis of our data storage also [1], so it is quite natural to evaluate the same methodologies. As their work required storing the full representation of the data, lossy methods for compression were not suitable. One of the key requirements was also streaming ability of the compression method, which differs from methods that operate on complete datasets or blocks of data. Each time series is compressed separately without any linking to other time series. Timestamps and values are also compressed separately, both using the information from previous values. However they are using different algorithms for timestamps and values.

Gorilla is a fault-tolerant in-memory time series database that functions as write-through cache for their older data storage. It does not handle long term storage and instead uses persistence only for warming up the caches in case of a failure. The open source version of this system is called Beringei[11] but is not suitable for our use-case due to the lack of out-of-order writes as well as providing only a precision of one second in the timestamps.

2.4.2 Chronix

Chronix[76] describes itself as a time series storage that is optimized for operational time series and achieves good resource consumption and fast query times. Chronix uses Apache Solr[7] to store and index the data.

Data is stored in data type they call a record, which stores a chunk of data points and metadata related to the series that describe it and can be used for query purposes. When storing the data chunks, Chronix can use lossy compression method they call semantic compression by Shafer et al. [113] which reduces the dimensionality of the data. Chunks are then serialized into bytes and in that process the timestamps are calculated into deltas between timestamps before being stored. A generic compression method gzip[120] is then applied to the chunk to reduce storage space requirements.

While the Chronix can fill multiple requirements of what we need, it is unable to process streaming data and requires chunks to be gathered before storing them or querying them. This would require us to build a secondary system to ingest data and to query from, increasing complexity of the solution.

2.4.3 Prometheus

While Prometheus[96] is not a time series database, but a complete monitoring solution, it is worth exploring here as it has gained a lot of traction in the market.

Prometheus newest data storage encodings are based on the Facebook's Gorilla paper with delta-delta compression on the timestamps and variable bit-width value encoding. Prometheus uses different delta of delta ranges when storing the timestamps and also the encoding of the chunks does not follow the paper's structure.

Prometheus is very different from what we are creating as it is not distributed at all and the work to get higher availability or scaled out are manual. It is also not designed for long term storage of the time series and has limited support for out of order writes because its storage is relying on the semantics of Prometheus' internal ingestion system. Those same semantics prevent it from being used as a generic time series database also, since it does not support anything beyond its internal metric fetching capabilities.

2.4.4 InfluxDB

InfluxDB is a open source time series database that is designed for high write and query loads. Clustering features are only available in double-write configuration in the open source version and more advanced clustering features require a commercial product. For storage it uses its own Time-Structured Merge Tree (TSM) storage engine, which is very similar to a Log-Structured Merge Tree (LSM). [64]

Compression process happens in the compaction phase, which stores the in-memory caches to the disk. Timestamps and values are stored separately in the tree, which allows the storage engine to use different compression methods for each type. Their compression method is adaptive and uses a combination of delta-encoding, scaling and Simple-8b. Timestamps are first delta encoded, then scaled if they have a common divisor of 10. After that, if they're in allowed range, the timestamps are Runtime-Length-Encoded (RLE) and finally Simple-8b[3] compressed.

For values, the encoding depends on the data type. With floating points, they use the XOR scheme from the Gorilla paper[92]. Integers are first ZigZag encoded[40] to remove signed bits and then compressed using Simple-8b. If all values are identical RLE is used to improve the compress ratio.

InfluxDB has interesting compression techniques with a very similar file structure and process to what Cassandra uses. One of the most complicated problems in the distributed system is however the distributed part and recreating one for InfluxDB instead of writing our own compression functionality is much more time consuming and error prone.

2.4.5 Cassandra based time series storages

Cassandra has long been a popular choice as a storage method for time series databases, with solutions such as KairosDB[70], Cyanite[27], NewTS[89], Blueflood[99], Heroic[54] and BigGraphite[26] to name a few. While all of them provide abstractions on top of the Cassandra suitable for time series querying, none of them have any support for compression beyond what Cassandra provides.

3 Compression

Data compression intends to reduce the amount of bits required to transfer and store data. Compressors can be both lossless and lossy, where lossless compression techniques allow to decompress the exact representation of the original data. Lossy compression algorithms on the other hand rely on the data reduction techniques by down sampling or extracting patterns such as seasonality from the time series. They're based on the concepts of signal theory in most cases [32] and because operational time series are not just sequences of random data and can be sparse in nature, they can be effectively compressed with data reduction techniques such as Fourier transform (FFT). This gives us a high level trend of the time series, but fails to capture all the anomalous events and spikes in the data.[105]

Operational time series often have anomalies and spikes, and their patterns can drastically change in short period of time. As detection of anomalies and spikes are important for multiple common queries in the monitoring applications, the lossy compression methods are not usually suitable for this purpose. [105] [32] [97]

All data compression methods have at least two components, a model and an encoder. Model estimates the probability distribution, while encoder chooses shorter codes for more likely symbols. An optimal solution for coding uses $log_2 1/p$ bits per symbol which has propability of p. There are efficient and even optimal solutions to coding, but no optimal modeling has been proven to exist.

3.0.1 Dictionary based coding - LZ77 and LZ78

Lempel and Ziv introduced the first universal algorithm for sequential compression in 1977[133] and 1978[134].

The algorithms define a rule for parsing strings and symbols from a finite alphabet into words of bounded length and then maps these words sequentially into decipherable code words of fixed length using the same alphabet. In other words, it replaces multiple instances of the same data by referencing a single copy of the data.

Both algorithms called LZ77 and LZ78 by their release years are based on the incremental parsing. Difference between them is that LZ77 uses a sliding window approach of previously seen data as a fixed size dictionary while LZ78 uses a dictionary which size may increase unbounded. The LZ77 variant has been shown to reach compression ratio of source entropy as the sliding window size approaches infinity[130].

All the current popular generic compression methods are based on the ideas in LZ77.

3.0.2 Huffman coding

Minimum redundancy coding[60], also known as Huffman coding, is an important algorithm for lossless data compression and entropy coding. It maps a set of alphabets with a given frequency to minimal codes, sorting them in a way that more frequent the occurence of the code word, the smaller the assigned integer prefix. Given minimal possible code word lengths, the Huffman coding can be proven to be optimal prefix

code[60], but this does not mean it is optimal among all lossless compression data methods.

Huffman code is constructed by taking two elements with lowest weight as the nodes of the tree and then iterating this step upwards in the tree, until all the symbols have been encoded to tree. Then following the tree we can construct the code words for each element.

3.0.3 Arithmetic coding

Huffman coding performs optimally when symbol probabilities are power of 1/2, but otherwise can take up to one bit extra per each symbol. In most cases, the probabilities do not match this expectation. Arithmetic coding[106], also known as range coding, has no such issue, as it does not have the restriction of mapping each symbol to integral number of bits and as such provides better compression possibilities.[129]

Arithmetic coding is based on the idea of encoding a whole message to a floating point value between [0, 1[. Range of the message is the entire interval, but the range is narrowed as more symbols are encoded. The more likely symbols reduce the range less, thus adding less bits to the message. This allows to generate a unique encoded sequence without generating a code word for each.

3.0.4 Asymmetric Numeral Systems

Modern day compressors uses either Huffman or arithmetic (range) coding for entropy coding. Latter has the ability to reach higher compression ratios in most cases, but the tradeoff is compression speed. Asymmetric Numeral Systems (ANS) by Jarek Duda is intended to get the best out of both codings. [37] It has similarities to range coding, but instead of coding a single range, it spreads those ranges across the whole interval, thus requiring only one state to be maintained instead of two. ANS uses random generator to initially distribute the symbols with assumed statistics and could in theory use a hash to initialize the random generator for encryption.

3.0.5 Run-Length Encoding

Run-Length Encoding (RLE) is based on the idea of replacing long repeating sequences of a symbol with the count of the repetition and the actual sequence. For example, a sequence of *AAAABBB* could be written as *4A3B*. It is a very simple algorithm that works effectively for certain types of data.

3.0.6 Delta coding

If the stored integer arrays are in a sorted order $x_1, x_2, .., x_n \leq x_{n+1} \forall n$), it makes sense to store only the difference between the successive elements and the initial value. The difference between values is often smaller than the original integers. This is called differential or delta coding and is a simple form of prediction where we

assume the previous value is close to the subsequent value. In this paper the term delta coding is used.

To calculate the original values, we can reconstruct the original array by the means of a prefix sum $x_j = x_1 + \sum_{i=2}^{j} \delta_j$. The downside is that random access to an integer requires summing up all the previous deltas.

3.1 Generic compression algorithms

3.1.1 LZ4

LZ4 is a lossless compression format derived[81] from the LZ77[133] with fixed byte-oriented encoding. It focuses on the simplicity of implementation with high compression and decompression speeds.

A compressed block is built on sequences where each sequence is a block of literals. Each sequence starts with a token that indicates the length of the literals. The literals themselves are uncompressed bytes copied from the original stream. After the literals there is an offset and match length which tell the decompressor which bytes to copy from the literals. It is possible to copy more than available buffer by processing the request again after each decompressed byte thus copying the same data multiple times. After decoding, next sequence is read.[104]

3.1.2 Deflate

Deflate is a lossless compression algorithm that uses a combination of Huffman coding and LZ77. It was originally designed by Phil Katz for PKZIP[93] and later released as RFC1951[33]. It is a patented compression algorithm, but can be implemented in a manner that is not covered by the patent as described in the RFC.

A compressed data is built of blocks, which can be either compressed or uncompressed. If uncompressed, the maximum size of a block is 65535 bytes, otherwise the size is arbitrary. Huffman coding is independent for each block, but the LZ77 algorithm can use input from previous blocks also.[33] Compressed blocks are first encoded with LZ77 and then with Huffman coding. Depending on the block compression code, the tree is either transferred with the data or it is from Deflate specification.[1]

3.1.3 Zstandard

Zstandard[135] (Zstd) is a fast real-time lossless compression format intended to replace most deflate use-cases. It is built with the idea of performance-first, using more memory than zlib, allowing parallel execution and using branchless algorithms. [115] It is intended to be a good generic compression algorithm targeting multiple use-cases. There is also a specific training mode for smaller files by using a pregenerated dictionary, which should speed up compression/decompression of small files and to improve compression ratio.

Underneath Zstandard uses LZ77 and for entropy coding Finite State Entropy (FSE), which is a new generation entropy coder designed for modern CPUs. It is a variant of ANS that precomputes coding steps into tables and uses only table

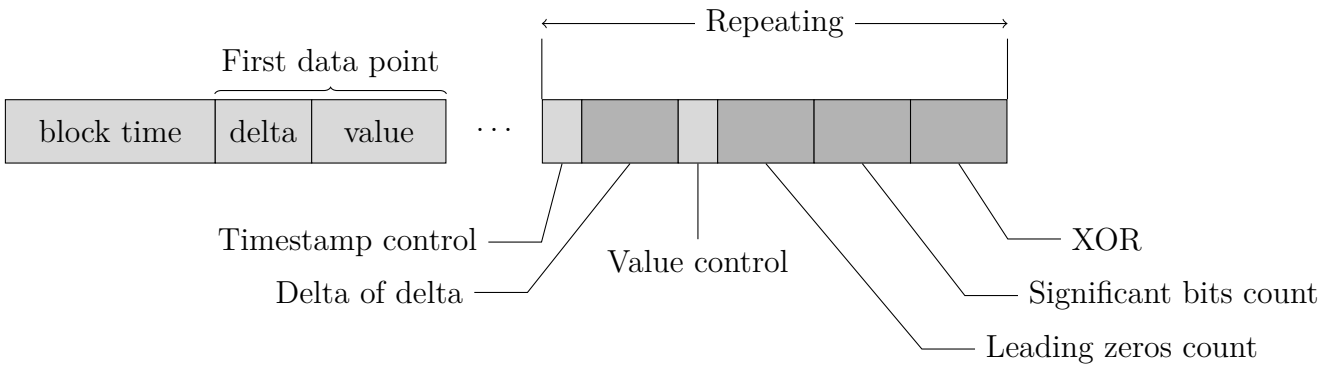


Figure 1: Structure of Gorilla encoding, blocks in darker grey are optional

lookups, shifts and additions to do the coding. This type of variant is also called tANS (table ANS).

3.2 Gorilla compression

To our knowledge, the Gorilla compression scheme is the only one that has been specially designed for the monitoring time series data. While it does not have any new inventions in terms of algorithms and data coding, it is an example of powerful modeling to target certain scenario. That scenario happens to be the one we are trying to solve also, which makes it even more interesting and important for our experiments. The encoding format supports streaming, which can be seen from the layout of the data as shown in Figure 1. This means a single binary block is also self-contained as it includes all the necessary data to reconstruct a time series.

3.2.1 Compressing time stamps

When Pelkonen et al analyzed the time stamp data stored in the Facebook's operational data store, they noticed that vast majority of data points arrive in fixed intervals. Sometimes the interval might drift small amounts to direction or another, but the window of drift is very small. Instead of storing the time stamps in full detail, they use the information of fixed intervals in the compression. Previous values are used as predictors for next ones and data is compressed in delta-of-delta. Variable length encoding is used in storing the time stamps and the implementation uses ranges that were selected based on the sampling of the real data from production systems and finding the best compression ratio. For example, they noticed that majority (96%) of the delta-of-deltas were identical to the previous one in their dataset.

The algorithm works as follows, given a block timestamp t_0 and the first stored timestamp t_1, a delta of $t_{-1} - t_0$ is calculated and stored with X bits. The amount of bits in the delta limits the maximum size of the block. After that, the sequence in listing 1 is followed until the block is closed or the timestamps are beyond the maximum addressable limit.

Algorithm 1 Encode a timestamp in Gorilla stream

$\Delta \leftarrow (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$
if $\Delta == 0$ **then**
    $stream \leftarrow 0$
else
    if $\Delta \geq 0$ **then**
        $\Delta \leftarrow \Delta - 1$
    end if
    if $\Delta \geq -64$ **and** ≤ 64 **then**
        $stream \leftarrow 10$
        $stream \leftarrow \textsc{LeastSignificantBits}(\Delta, 7)$
    else if $\Delta \geq -256$ **and** ≤ 256 **then**
        $stream \leftarrow 110$
        $stream \leftarrow \textsc{LeastSignificantBits}(\Delta, 9)$
    else if $\Delta \geq -2048$ **and** ≤ 2048 **then**
        $stream \leftarrow 1110$
        $stream \leftarrow \textsc{LeastSignificantBits}(\Delta, 12)$
    else
        $stream \leftarrow 1111$
        $stream \leftarrow \textsc{LeastSignificantBits}(\Delta, 32)$
    end if
end if

Because the 0 value is always catched in the case B, it can be used in the other blocks as a value. For this reason, if the value is positive, it is reduced by one in the compression phase and increased again in the decompression phase. There is an error[36] in the original paper where the values are [-63, 64], [-255, 256] and [-2047, 2048].

3.2.2 Compressing values

For value compression, Facebook's engineers looked at compression schemes of scientific floating point data which leverage the XOR comparison[92]. Scientific floating-point methods in[101] [79] are value-prediction-based compression algorithms for IEEE 754[63] standard (see Figure 3) which is a 64-bit floating-point format that uses 1 bit for sign, 11 bits for exponent and 52 bits for mantissa. In short, both methods work by first predicting the next value in the sequence and then comparing it to the real value and compressing the difference. If the predicted value is close to the real value, then the sign, exponent and first few mantissa bits will be the same [101].

To store the difference, [101] uses XOR, which is a reversible operation that turns identical bits into zeros and the XOR result is expected in these cases to include substantial amount of leading zeros and if the prediction is close to the true value, the difference can be stored using few bits. And because many CPUs support machine instructions for counting leading zeros, this can be used to accelerate the algorithms.

In the real world data set used to model, they discovered that values do not change a lot between neighboring data points and this allowed them to replace the computationally expensive predictors in [101] and [79] by using the previous value in the time series as a predictor. In their data, a total of 51% of values do not change at all from previous value, while 30% can be compressed with identical leading and trailing zeros count as previous value. Further, many values they store are actually integers, which compress especially well because location of set bits are often the same in the series. [92]

First value in the series is stored uncompressed (using 64 bits), while the next values follows the algorithm described in Listing 2.

3.3 Integer compression techniques

While there is large amount of integer compression techniques, most of them are only interesting in a historical sense as their performance and compression ratio lags behind some of the newer options[77][136][112]. For that reason, we will only look at the newer integer compression methods that are designed for the modern CPUs.

Structure of compression formats is built on two layers, control patterns and data parts. [77] Data parts store the encoded data, while the control patterns describe the information that is required to parse the structure. Encoding methods can be divided to sub categories based on their data alignment strategies.[117][77]

Algorithm 2 Encode a value in Gorilla stream

1: $x \leftarrow \oplus(input[i-1], input[i])$
2: **if** $x == 0$ **then**
3: $\quad stream \leftarrow 0$
4: **else**
5: $\quad l[i] \leftarrow \textsc{LeadingZeros}(x)$
6: $\quad t[i] \leftarrow \textsc{TrailingZeros}(x)$
7: $\quad x \leftarrow \textsc{ShiftRight}(x, t[i])$
8: $\quad m \leftarrow 64 - l[i] - t[i]$
9: $\quad$ **if** $l[i] \geq l[i-1]$ **and** $t[i] \geq t[i-1]$ **then**
10: $\quad\quad stream \leftarrow 10$
11: $\quad\quad stream \leftarrow \textsc{LeastSignificantBits}(x, m)$
12: $\quad$ **else**
13: $\quad\quad stream \leftarrow 11$
14: $\quad\quad stream \leftarrow \textsc{LeastSignificantBits}(l[i], 5)$
15: $\quad\quad stream \leftarrow \textsc{LeastSignificantBits}(m, 6)$
16: $\quad\quad stream \leftarrow \textsc{LeastSignificantBits}(x, m)$
17: $\quad$ **end if**
18: **end if**

Table 1: Encoding modes for Simple-8b

Selector	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Integers	240	120	60	30	20	15	12	10	8	7	6	5	4	3	2	1
Bits per integer	0	0	1	2	3	4	5	6	7	8	10	12	15	20	30	60
Wasted bits	60	60	0	0	0	0	0	0	4	4	0	0	0	0	0	0

3.3.1 Variable Byte, Byte-aligned

Variable Byte is a byte-aligned method[77][128] that uses seven bits to store the data and the eight bit is used as a control pattern for code length. The least significant bit is set to 1 if the sequence continues and 0 in the last byte. Variable bytes are not generally the most optimal way of compressing integers[77][128] but they are reasonable effective, especially in the byte aligned storage if the amount of integers is small [128] or the data is not highly compressible[77].

While the classical variable byte uses single bit in each byte as descriptor, several variants such as varint-G8IU[117] and Group Varint[29], also known as varint-GB, use different arrangements for faster processing.[77]

3.3.2 The Simple family, Word-aligned

The Simple family of codecs encodes a variable amount of integers to a fixed number of bits, using 32-bit word length in Simple-9[4] or Simple-16[131] and in 64-bit word in the newer Simple-8b[3]. The only interesting case here is the Simple-8b because we need to encode 64-bit long integers and Simple-8b is the most performant alternative[77].

As seen in Figure 2 the first 4 bits of the 64-bit word indicate the encoding mode and the remaining 60-bits are for the data, in which all the integers are encoded with the same number of bits. It is a greedy algorithm, trying to fit all the integers using smallest possible number of bits as possible starting from the lowest number. Due to the structure of all Simple-family codecs, the Simple-8b is able to store an integer of max size 2^{60}, while the Simple-9 and Simple-16 are only able to store 2^{28}.

For example, if the integers can be encoded using 6 bits each, then selector 7 (0111) is written to the first 4 bits and 10 integers are encoded. The Table 1 shows all the possible combinations.

Simple-8b provides an advantage over all the other solutions by being the only solution that does not split over code words[3]. All the information required to decompress a set of compressed integers is in the same word and no external information is required.

3.3.3 Frame-Of-Reference

Binary packing or Frame-Of-Reference (FOR)[46] operate on an array of values which are partitioned in to blocks (such as 32 integers or 128 integers). The idea is to find smallest value m and the maximum value M as the range from the partition of integers and encode this range. The smallest value is encoded as 0 and the other

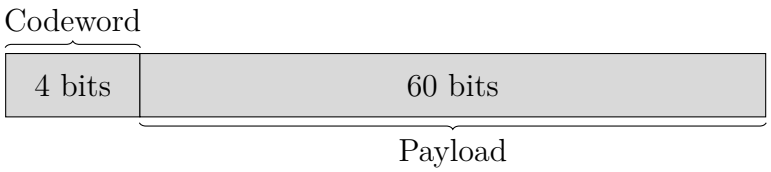


Figure 2: Structure of Simple-8b compressed word

values are encoded as offsets of this smallest value m. Each integer can be stored in the $log_2(M - m + 1)$ bits. Ahn and Moffat[3] called their version PackedBinary while Delbru et al. [31] used the term FOR of their 128-integer version and AFOR-1 for their 32-integer version.

For example, consider values [1050, 1056, 1060, 1078]. The range of values is [1050, 1078] and we can subtract 1050 from each value to encode only the difference. That gives us a sequence of [0, 6, 10, 28] in which we can store all the values with 5 bits each. We also need to store the original value of 1050 and 3 bits to encode the 5 bit requirement. Bit packing functions are very fast[77].

There are also versions with variable-length blocks. These require a small overhead to store the block length, but in return can optimize the block length for best compression ratio. Delbru et al. applied[31] a greedy algorithm for this problem in their version can called it AFOR-3. Silvestri and Venturini used a dynamic programming to approach the same issue to find the optimal partitioning and calling their version Vector of Split Encoding (VSEncoding)[114].

3.3.4 Patched coding

Zukowski et al.[136] noted in their paper that in practice data sets can be skewed in value distribution and this causes the binary packing to perform badly in terms of compression ratio since FOR cannot cope with outliers in the data. For example, storing integers [1, 2, 3, 65535, 4] requires 16 bits to store each value. To solve this, they proposed patching to store outliers as exceptions so that the range of values is strongly decreased and called this encoding method PFOR.

The input array is partitioned into smaller blocks called pages which have fixed maximum size and single bit-width is used for the entire page. The exceptions are stored in separate location from the rest of the values. To encode the values, we use bitpacking by encoding either the value itself if it fits the bit-width constraints or the offset pointing to the next exception. The offset is stored as the delta of current exception and next exception and then subtracting one. Consider the following example of integers:

110, 1, 1, 10, 10000, 1, 1, 10, 100111, ...

Assume that the bit-length has been set to 3, so we have exceptions in positions 4 and 8. The offset is 8-4-1=3, therefore we still store '11' in place of the value. The end result is:

110, 1, 1, 10, 11, 1, 1, 10, ...

To find the first exception, in front of the encoded values there is a 32-bit header which contains two markers. One for the location of the first exception and second marker is for the location of first exception in the encoded array. This allows us to find the first exception at position 4, read the value '11' and detect that next exception is 3 indexes ahead.

3.3.5 Improvements to PFOR

The PFOR does not compress the exception values and with a large amount of exceptions it is possible to overflow the available space. Yan et al. [131] proposed a two variations called NewPFD and OptPFD to improve compression ratio. Instead of using a single bit-width for the whole page, both OptPFD and NewPFD use a bit-width per 128 integers and all the exceptions are stored with Simple-16. The difference between these two variations is that NewPFD picks a bit-width such that no more than 10% of the values are exceptions while OptPFD picks the bit-width that maximizes the compression ratio.

While NewPFD and OptPFD reach a higher compression ratio, their performance is lower than the original PFOR's. As a trade-off, Lemire at al.[77] proposed a method called FastPFOR. Like PFOR it stores all the exceptions at the page level using compression, but chooses the bit-width on per block basis like NewPFD and OptPFD.

3.4 Value predictors

Lipasti et al.[80] introduced the term "value locality" to indicate the likelihood of a recurring value in storage applications and as result they can be predicted. Value sequences can be categorized to three different sub-categories, constants, strides and non-strides[109]. Constant sequences are the simplest form, repeatedly producing the same result, but this occurrence is still very common in systems[80]. Stride values are sequences which are separated by constant deltas, such as a sequence of 1, 2, 3, 4, .. the delta can be a positive or negative number. The non-stride category is then all the rest sequences. Common pattern is also seeing a repeated pattern of stride and non-stride sequences[109].

From these definitions the predictors can be modeled in two ways, *computational predictors* that make predictions by computing a function on the previous values and *context based predictors* use the history of recent values called *context* to predict the next value.[109]. These predictors use simple and fast operations for their algorithm and as such are good candidates for compression purposes.[101]

3.4.1 Computational Predictors

Computational predictors are based on the idea of running a function on top of the previous value that was extracted. A last value predictor was introduced by Lipasti

et al.[80] which in a very simple form predicts that if the previous value is v then the next value is also v. Gabbay et al.[44] introduced a stride predictor which assumes that the underlying values are built on a stride-pattern. A delta is calculated between the previous value and the preceding value and then added to the preceding value to calculate the next value.

3.4.2 Context Based Predictors

Finite Context Method (FCM)[109] predicts the next values by using a finite number of previously-seen values. Unlike the computational predictors, FCM does not assume any relation to the previous value. The implementation uses a two level table, where the first level stores the recent history of values and the second level stores all the possible patterns that can follow it. A counter is placed next to each value that calculates the occurrences of the value following a certain context and the counter is updated after a value has been seen. The value with the highest count is then used as the prediction.

The length of the history is called order of the FCM. FCM predictor can be used to replace both stride predictors and the last value predictors, but the learning period is a lot longer[109] since it must first fill the history table before it starts to make predictions. To improve the efficiency of the second level table each historical value should map to a different entry in the next level and to accomplish this a hashing function is used, such as XORing the different values together.[108] The hashing also allows the history to be computed incrementally by using the historical values and the new value and storing hashed history in the first level with the correct values in the second level only.

Goeman et al. [45] showed that FCM predictors have issues when dealing with stride patterns or repeating stride patterns, causing the tables to fill and leaving little space for any other predictions. Even worse, since each stride pattern at length n will require n elements in the second level table, any stride pattern longer that is longer than the size of the FCM table will destroy itself. To prevent these issues when dealing with stride patterns, they proposed a modification to the FCM called Differential Finite Context Method (DFCM). To fix the issue with stride patterns the modification is to store the differences between the values instead of the values. With that, all the stride patterns become constant patterns and the DFCM can correctly predict even the stride patterns that have not been seen yet. DFCM does not use the last value as the indicator for second level index to avoid further issues with the stride patterns. Instead, it combines FCM and stride predictors access methods by using program counters to read the last value and the hashed history from the first level of the table and searches the correct index from the second level of the cache and the difference is added to the last value to predict the value.

The downside of DFCM is that non-stride patterns might interfere with each other in the DFCM which does not happen in the FCM and that DFCM requires larger first level table to store because it needs to store the last value. There is also a small amount of extra processing requirement due to subtract and add processes, but in most cases this is not a problem for overall performance. The benefit is better

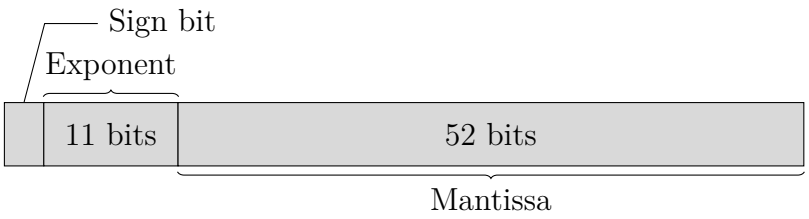


Figure 3: Structure of IEEE 754 double precision floating point

prediction accuracy with smaller tables as FCM requires huge tables (up to 2^{20} entries) to match the prediction accuracy of DFCM.[45]

The DFCM can suffer from bad selection of hashing function, for example on byte loads. [13] Thus, most implementations that use DFCM have different hashing functions depending on the target payload to improve prediction accuracy.

3.5 Double-precision floating point compression algorithms

To understand why lossless compression of floating points is difficult[101][17][43][47][15][110] we have to look at how the values in the IEEE 754[63] standard are stored. All the values are made of a sign bit s, 11 exponent bits e and 52 mantissa bits m (Figure 3).

Since the sign, the exponent and the top mantissa bits take a large amount of the bit space, we can expect the XOR result of the two values to be close to each other.

Methods available with integer compression are not usually applicable to floating points. As an example, delta coding could be used in a sorted integer list to reduce the amount of bits required to be stored. With the floating point values, this might cause underflows with loss of information and the end result might not have less bits to present the information[41]. Instead, a XOR operation is used to compare two different values. XOR is a logical reversible operation that outputs false if bits are identical. This works well for values that are very close, such as 10.000000000000000 and 10.000000000000001 as shown in Figure 4. This is because the exponent, sign and the top mantissa bits take a large amount of the bit space.[79]

0100000000100100000000000000000000000000000000000000000000000000	10.000000000000000
0100000000100100000000000000000000000000000000000000000000000001	10.000000000000001
0000000000000000000000000000000000000000000000000000000000000001	$\oplus$

Figure 4: XOR example of two close floating point values

However, this is not true for all the values, such as *1.0* and *1.1* as we can see in the Figure 5. Here we need to store the XOR with 49 bits and there is only 15 leading zeros.

It is also common to store integer values as floating points[79][92][50][123] and they might exhibit same problems. As an example, while storing values *1* and *2* in integer format will create a XOR with 62 leading zeros and 2 bits of information, storing the same in floating point format *1.0*, *2.0* will result in a XOR with only 1 leading zero followed by 11 set bits.

0011111111110000000000000000000000000000000000000000000000000000	1.0
0011111111110001100110011001100110011001100110011001100110011010	1.1
0000000000000001100110011001100110011001100110011001100110011010	$\oplus$

Figure 5: Another example of floating point XOR

Engelson et al. [41] treated the floating point series as time series that are polynomial in nature. The paper proposes a method where delta calculations are used to subtract precedent value from the next value, but instead of using floating point arithmetic the operation is done with the integer representation of the floating point. Data was then truncated to align with certain bit length by removing leading zeros. Klimenko et al.[71] processed the Laser Interferometer Gravitation Wave Observatory (LIGO) time series data using floating point subtraction and null suppression on the resulting data.

While previous work concentrated on compressing consecutive values, Ratanaworabhan et al.[101] searched for a more general solution to the problem by trying to identify recurring difference patterns and using a modified DFCM predictor to forecast the next value. The predicted value is then XORed with the actual value and the resulting prediction error is then stored. This should result in large number of leading zeros, if the prediction is close to correct, which are then encoded using 4 bits for leading zero count resulting in half-byte granularity.

Lindstrom and Isenburg[79] concentrated on the 2D and 3D structured floating points in their compression algorithm by using a Lorenzo predictor[61] and storing the residual. Burtscher et al[15] used two different predictors, FCM and DFCM, in their FPC algorithm and encoding the difference and which predictor was used. The paper considers FCM and DFCM as the best combination as a paired predictors, not necessarily the best predictors for all the data sets. FPC is byte-aligned for performance reasons and processes always two doubles at once. FPC has also got improved versions, a parallel version called pPFC[16] and a self-tuning version which changes the predictor's hash function based on the input and predictor accuracy, called gFPC[14].

Schendel et al. [110] looked at the problem of lossless compression of floating points from a different perspective. Instead of trying to improve existing algorithms to compress better, they created an ISOBAR preconditioner to analyze the compressibility of the data and to partition it to compressible and incompressible segments. After that the existing lossless compression algorithms can be used on the compressible parts to improve the total throughput and compression ratio.

Gomez et al.[47] took a similar approach as ISOBAR, but tried to find a bijection between the original values and a set of masks to reduce the entropy of the floating point data set. Like in the ISOBAR, they improved the idea by detecting which parts of the floating point structure benefits most from the masking technique and only apply the optimization to those byte parts. Burtscher et al.[17] have proposed a method of using multiple different algorithms and trying to compose a best one for each data set.

3.6 Implementing compression algorithm

With modern CPUs, for a CPU-efficient (de)compression, following principles should be followed to create vectorized compression and decompression algorithms[136]:

1. Arrays of values should be compressed/decompressed in a tight loop
2. if-then-else inside the loop should be avoided
3. The loop iterations should be kept independent

First rule allows the compression to happen between CPU and cache boundary, instead of accessing main memory or I/O storage interface.[136][77] The second rule comes from the fact that modern CPUs can execute instructions in long pipelines with several instructions simultaneously and out-of-order. If-then-else can cause control hazards, and although modern CPUs have branch predictors to predict the outcome of the branch, a mispredicted branch will require flushing the pipeline and starting over. Last rule allows more efficient out-of-order work as work is not dependent on previous iterations. [136]

Another approach to pipelining is to use Single Instruction, Multiple Data (SIMD) instructions available on modern CPUs to parallelize computations[78][111], which is is also called vectorization. SIMD abilities often require modifications to the algorithms to avoid data alignment or organization issues and as such are often not automatically applied by the compilers.

With fast compression methods, main memory might become a bottleneck and to prevent this, differential coding and decoding makes sense to be done in place. To reduce the cache misses and to avoid bandwidth from becoming an issue, larger arrays must be broken into smaller arrays for compression. Reducing cache misses by processing smaller blocks can lead to significant performance improvements. [77]

4 Research material and methods

While time series themselves are increasingly important subject and extensively evaluated, most of the work is based on the ideas of lossy compression and finding material for lossless time series compression is limited. This includes very specific research towards compressing only certain types of data, such as smart grid [38][2] and other sensor networks. Sadly, this type of research did not usually provide us with information regarding the monitoring data setup and in many occasions we had to use other available software and industry knowledge to validate our plans. Far more information is available from columnar databases, where the use of compression is a very well researched topic in projects such as MonetDB[119].

The used compression algorithms in columnar databases themselves are not often interesting, but by thinking time series data as a sequence of numbers leads us to the floating point and integer compression algorithms. Combining the data storage information from columnar databases with the algorithmical information of numerical compression gave us possible steps in approaching the issue. We also paid attention to the generic compression algorithms and how they use more advanced predictors and coding to see if certain properties from them could be acceptable to our solution.

4.1 Experiments setup

When benchmarking performance, we run the same scenario for thousands of times and recorded the results using HdrHistogram[53]. The HdrHistogram is initialized as expanding with a set of precision, thus allowing enough data points to be gathered to get reliable results. HdrHistogram adds negligible overhead when recording the data and as such does not affect the performance runs in a significant way. If the algorithm is built on the JVM, enough runs are run to ensure that the JIT can optimize the runtime code for best performance.

Data set parsing and memory allocation for parsing processes are not counted towards the timing, but are instead done before the actual execution. If the allocation is internal to the algorithm, it is counted to the execution time like it would be in the real world use case. This is something each algorithm does on their own. When possible, we attempted to measure the allocation impact to the end result and will share our findings in the observations part.

Test setups differ slightly for each data set, although recording the results is done with the same tools.

Zipf sample is generated using a rejection-sampling method for Zipf distribution, since this is a very fast method for creating large Zipf samples. These values were always benchmarked directly from the sampled array without writing them to the disk.

InfluxData comparison data was created by the tool from InfluxData and then sent to the Hawkular-Metrics instance backed by a single node Cassandra. All the tests read the data directly from the resulting SSTables.

Failed Openshift performance run data set was provided by the scalability team from Red Hat as SSTables from the storage. All the tests read the data directly from

the resulting SSTables.

Windows performance monitor data set was stored as CSV format from the perfmon application. It was then read by a custom Python script that sent all the data to the Hawkular-Metrics instance backed by a Cassandra. All the tests read the data directly from the resulting SSTables.

Bitcoin values data set was provided as a set of CSV files, which were then parsed by a custom Python script that sent all the data to the Hawkular-Metrics instance backed by a Cassandra. All the tests read the data directly from the resulting SSTables.

The data sets are described in more details in *Data sets* section. We performed the reading tests directly against the SSTables to reduce the amount of unknowns when testing exact benchmarks. All the SSTables were served from the page cache as there was enough memory in the system to remove any I/O bottlenecks from affecting the results. While the data loading used Python for the CSV scripts, all the performance and compression tests were done with a Java tool created for this occasion. For performance benchmarking, all the necessary data was placed to an array placed in the heap and not read all the time from the SSTables.

4.1.1 Hardware

These tests were run on a physical hardware without virtualization. The hardware in question used Intel i7-2600K processor, which supports instructions up to AVX. The system was equipped with 32GB of DDR3 running at 1333MHz and the tests were run from a Sandisk Ultra II 480GB SSD. No I/O bottleneck was noticed during the testing as all the data sets were served from a page cache.

4.1.2 Software

Name	Version	License	Source code
OpenJDK	1.8.0u131	GPL v3	hg.openjdk.java.net
LZ4 Java port	1.4.0	Apache License 2.0	github.com/lz4/lz4-java
Deflate (JDK8)	1.8.0u131	GPL v3	hg.openjdk.java.net
Gorilla	2.0.4	Apache License 2.0	github.com/burmanm/gorilla-tsc
Simple-8b	0.2.0	Apache License 2.0	github.com/burmanm/compression-int
FPC	-	Apache License 2.0	github.com/burmanm/fpc-compression
JavaFastPFOR	0.1.11	Apache License 2.0	github.com/lemire/JavaFastPFOR
Zstandard	1.3.1	BSD	github.com/facebook/zstd
Cassandra	3.11.1	Apache License 2.0	github.com/apache/cassandra
Hawkular-Metrics	0.28.2	Apache License 2.0	github.com/hawkular/hawkular-metrics

Figure 6: Used software

There was an existing implementation of FPC for Java[72] but it performed so badly in terms of compression speed that we decided to write our own implementation which is 2-3 times faster. This is to ensure that the comparison between FPC and other compressors are about algorithm and less about implementation details.

Since the PFor implementation by Lemire only supports 32-bit integer values, we provide results for it only if all the values in the tested range are smaller than the maximum signed 32-bit integer can support. Also, Simple8 does not support values larger than 2^{60}, so it too required excluding certain values. Larger numbers are common if measurements are reported in small units, such as nanoseconds.

We chose not to extensively test more generic compression algorithms as their results are often behaving like LZ4, Deflate or Zstandard, depending on the trade-offs they make. When encoding using generic compression method, we employed some of the entropy reduction techniques from the special algorithms to get more realistic results from them. For example, when compressing the timestamps, we first calculated delta-of-delta results for all values to increase the amount of leading zeros in each value. We provide a comparison to the more naive approach which uses an array of original values compressed with the generic compressor to show the difference.

We tested the Zstandard with multiple different compression levels, but opted to use level 6 for these tests. We found that this level gives a huge performance boost compared to slower levels, but there was no significant loss of compression ratio. Facebook opted to use level 3 in most of their use-cases[115], but we found that reduces compression ratio too much for our needs. For LZ4, we ran our automated tests using the Java port, but also verified that the results are comparable to the C version by Yann Collet.

4.2 Compression method evaluation criteria

This paper is seeking to find a practical solution for the data compression method in a distributed time series database. When evaluating the potential algorithms to use, we must exclude any algorithm that might be protected by patents as we ship our open source project with a patent free clause[50][6].

When evaluating the compression methods, the usual metrics are encoding speed, decoding speed and the compression ratio. The encoding/decoding speed is defined by the rate of processed integers or floating points, while the compression ratio is the ratio between compressed and uncompressed sizes of the dataset. While we could have measured the rate of bytes processed we opted not to use that, since we are interested in seeing how many data points we can compress in certain amount of time.

While algorithms can be evaluated on a theoretical level, in this paper we concentrate on the practical evaluation as that is the end result we are trying to improve. In performance evaluations we calculate the possible initialization costs per each block, including any allocations necessary by the compressor. In the compression ratio, we include all the necessary bytes to produce a package which is self contained and includes all the necessary information for decompression. To get comparable results, all the tests are done on a platform without virtualization and are repeated multiple times get a stable result. We want to push the error margin below 2% as we have used the same boundary on our QE for other performance tests. This allows us to evaluate the difference of performance to other improvements made in the

software.

As there is no industry standard for these measurements or anything comparable, the software used for benchmarking will be released to enable the possibility of recreating the environment later if some new usage pattern or compression algorithm emerges.

4.3 Architectural evaluation criteria

The architecture of the solution has certain limitations which the solution must fulfill that come from the restrictions of the runtime and the use of Cassandra:

- Hawkular-Metrics has no persistent storage, it can only save state to Cassandra
- Often ran in an environment that does not guarantee consistent runs (restarts and node jumps must be assumed as a norm instead of an exception)
- Solution should be idempotent until state is saved to the Cassandra
- Data sets cannot be assumed to fit memory when compressing
- Retention times are not limited and more data should not slow down the system

4.4 Data sets

We used both generated data sets as well as real world data sets to provide us with some background for our choices. Unfortunately, we could not find available public data sets for machine monitoring data which meant that the results are not easily replicable by external parties. We wanted to show results from as many edge cases as possible and for that reason each data set is quite unique compared to the others and do not test the same things. We have tried to push generated data sets to mimic certain behavior of monitoring data sets, while still giving them a purpose of testing something not catched by the real world sets. For our public data set, we picked something that is not monitoring data set, since we do allow storing other type of data also, even if it is not in our target for features.

4.4.1 Synthetic data

Anh et al.[3] discussed creating two sort of synthetic data sets, cluster data and uniform, for their tests of Simple family codecs when comparing to other integer compression techniques. While these synthetic data sets do not necessarily match with any known payload our application is going to encounter and as such bear little meaning in terms of comparison, they provide a solution to testing the correctness of

our solution. Lemire et al. used[77] generated these data sets in their implementation, but used the values in a sorted order for their tests. Since we wanted to exploit the unknowns of the time series, we opted not to use sorted data sets for our tests as it is not uncommon for a monitoring time series to experience anomalies and sudden spikes.

In our use-cases the delta encoding is a method often used to reduce the size of the stored data. We calculate the deltas of values when storing both the timestamps as well as counters. The distribution of delta values follow a power-law, a positively skewed distribution, meaning that most of the values are on the left side of the distribution. At the same time, values in a highly skewed lists usually also appear to be clustered meaning we get long sequences of similar values. To create a dataset that exhibits these properties, we used a Zipfian distribution of values with sufficiently high skew. Zipfian distribution is a power-law probability distribution with integer values. The redundancy of Zipf's distribution increases logarithmically with the sequence length and thus allows approaching entropy with longer sequences.[42]

We used large Zipfian distribution for automatic verification for the correctness of our compression implementations, since we gain both long sequences of similar values as well as occasional larger values, thus hopefully targeting all parts of the encoding and decoding sequences. We refer to this data set as *Zipf sample.*

Second synthetic data set is from a tool created by the InfluxDB team to compare different time series solutions to their product. It is generated by using a defined random seed to always create the same values and it tries to mimic some patterns such as disk I/O, CPU usage and disk usage. The InfluxData comparison creates 87.1% of metrics as floating point numbers and 12.9% as integers. Due to the low number of integers, we opted to use this data set as an example of writing everything as a floating point data point. This is pretty common case in many software, where the collector agent does not know the real data type and thus has to make a guess. A floating point is a safe guess as it can store larger numbers than the maximum integer number, losing some precision in certain cases. This data set is referred as *InfluxData comparison*.

4.4.2 Real world data

Windows performance monitor. As an external data set, we wanted something that is not collected by our own tools. This data is collected by Windows' internal monitoring collector and was stored to a CSV file, which we then parsed with a small Python script (see Appendix B) and stored it through Hawkular-Metrics to Cassandra. The data set contains 19596 time series with a total of 209110368 data points. All of the data is stored as floating points as there is no type information in the CSV file. Timestamps collected were in milliseconds with 80% of the values being same as previous one.

Failed Openshift performance run. We selected this data set as it shows an interesting case where data is not continuous, but has runs of equivalent values followed by large value and then returning to normal repetitive flow. This is because the data set contains was produced in a situation where some data pushes failed and

were then back buffered in some cases. In the timestamps section it means long runs of 0 followed by a large number when the pushes resumed. In the integer and floating point values, we see same value repeated often as it is the same container that was started thousands of times. Due to the missing pushes, a single series of data can range from 3 data points to 240 data points (30 s interval was used in this test).

The data set is interesting as it exploits unpredictability in the compressed data and makes prediction much more difficult. It also favors RLE encoding over successful prediction schemes because of the continuous runs of same value. 78% of all timestamp delta-of-delta values were 0.

Bitcoin values. To show how our compression might work with something not related to the monitoring data, we picked up a public dataset of bitcoin values.[12] This dataset results can be replicated by everyone as the information is available. These results are in one minute interval from several bitcoin exchanges, between 2012 and 2017. The data columns are open price, high, low and closing prices of the day as well as volume in BTC and in indicated currency (EUR or USD, depending on the exchange). In total, there are 77149926 values split to 56 different series.

5 Design and implementation

In this section we describe our implementation and what parts we were required to build to get it working. We built our own compression algorithm implementations as well as modifications to our own software. The link to the source code for the whole solution can be found from Appendix B.

The Cassandra based processes are only available as part of the Hawkular-Metrics solution no parts of it were published as separate modules as the implementation was quite tightly integrated to our small core. However, other software could use our core-modules and implement their own interfaces on top of them as we have separated all the interfaces to their own libraries. The core library without the dependencies is only 238kB, including all the monitoring hooks and exposes all the functions as Reactive Observables with configurable asynchronous behavior.

5.1 Compression algorithm implementations

We implemented three different compression algorithms, one (FPC) which is used only for the comparison for this paper but the two other (Simple-8b and Gorilla) are available in Hawkular-Metrics. We used these compression implementations to do the actual compression in our Hawkular-Metrics solution, but also packaged them as separate libraries to allow use of them in other applications. We implemented the compression algorithms as there were no suitable implementations available and we used the occasion to implement better versions of them instead of only converting available solutions from other languages to our solution.

5.1.1 Simple family implementations

The low performance of Simple-8b is because it is difficult for JVM to optimize correctly in the compression phase. In the compression, the compiler is unable to determine how many loops are required since the steps depend on the previous data. Thus, no loop unrolling can be done and no JIT optimization to avoid unnecessary branches to be evaluated. And, inside the loop we have multiple branches with a requirement to know previously processed values. This prevents the CPU from actively executing commands out-of-order and as such it is starving for things to execute. The actual encoding phase is quickly done once we have determined the correct amount of bits to use per integer and how many integers to compress.

Original implementation uses greedy approach by trying first to check can it fit the maximum amount of integers to the next word. It does this by looping over a given amount of next integers and checking if any of them requires more bits to store than the current selector could fit. It does this until it can find a solution that will work and then compress the word using that found solution. We took another approach in our implementation (see Appendix B for link to source code) by never iterating the same integer again. We check how many bits are required for the next integer to be stored and then calculate if we still have bits available in our current selected selector using a table lookup. If the next integer will not fit our currently

selected selector, we check if the new selector would fit as many integers as currently we have on our queue. If there is enough space available in the selector chosen by the current integer's number of meaningful bits, we use that selector. Otherwise, we use the previous selector value.

It however means that the current selector might not be full and as such we need to roll forward the selector word to something that will be completely filled as Simple-8b cannot store partial words. Lets take an example of storing the values $5, 3, 4, 2, 1, 6, 7, 9, 1, 8, 5, 32, 5, 214, 6, 1, 123, 12, 0$. We would read the first value 5 first and notice that it requires 3 bits to store. We would use a selector 4 that can store 20 integers each using 3 bits. Next, we read 3 and notice that it can be fit to 3 bits and we up the count of used bits to 6 now. We can read the following values until we hit 9, which requires 4 bits to store. At that point we must check if the 4 bits selector can store enough values for us. It can store 15 values and has 60 bits available space for values to be stored to. We use that selector and notice that we are now storing $8 * 4 = 32$ bits. We can keep selecting until 32, which would require 6 bits to store.

At that point, we can notice that 32 would require 6 bits to store each value and since we have 12 values (32 included) that would require 72 bits. But since we only have 60 bits available, we have to stop now. Thus, we mark the previous bits required, which was 4 bits and we select 11 values. Now, comes the second part of our selection. Since we now have 11 values with 4 bits each, we only use 44 bits out of 60. Since this is not an acceptable option in the Simple-8b, we must up the amount of bits required to store the values. Thus, we try first by using 5 bits which results in 55 bits being used. As that is not enough, we have up the bits once more to a value of 6. We now do a table lookup and notice that 6 bits can store 10 values and we proceed to encode the next 10 values using 6 bits each. We now roll our counter forward by 10 numbers and in the next loop we start from the 11th value 5.

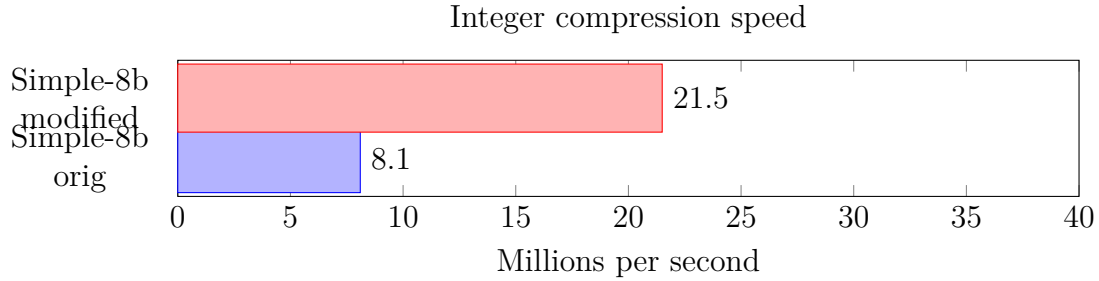


Figure 7: Simple-8b compression speed with algorithmic change

The effect of our modified algorithm for encoding can be seen in Figure 7, which confirms that we more than double the compression speed compared to the original solution.

In the decompression code we wanted to allow loop unrolling and fast unpacking and as such we used Java's switch clause for the read selector. This is actually a small table lookup in the final compiled code and allows two optimizations for the JIT, loop unrolling and loop optimization to avoid unnecessary index checks. We also made the function very small by writing each decoding block to their own function and

combined with the loop optimizations this removes the need for any branch prediction code. Same approach was taken by Lemire in his C++ implementation[77].

We also implemented an RLE variant of Simple-8b, which uses modified selectors. The selector 0 and 1 in the original Simple-8b have been removed and replaced. Instead, selector 0 is reserved for end-of-stream marker and rest of the selectors are moved forward by one. With these changes, selector 15 is available, which is used for RLE encoding of smaller values. With the selector 15, the first 28 bits are reserved for the count of repeating values, while the least significant 32 bits are reserved for the actual value. This scheme is originally by Lemire, but we use our optimized algorithm here also to compress the values.

5.1.2 Gorilla implementation

We have implemented two different versions of the Gorilla algorithm. First one followed the paper's algorithm quite closely with only a small difference in the size of the block as we used millisecond precision instead of second precision like the Facebook's system. In original paper, the block size is measured with 14 bits, which is enough when using second precision for up to 4 hours, but with milliseconds we would run out of space after only 16 seconds. To support up to one day with millisecond precision, our implementation uses 27 bits for the block size. Otherwise it was algorithmically equal to the paper and other implementations such as go-tsz[34]. However, it is the only such implementation done in Java.

Original implementation does not address all the requirements we have for our storage and some of the solutions were not optimal. For this reason, we implement a second revision, which improved the algorithm with the changes being divided to two different categories, storage and performance. As our use case stores a lot of 64 bit integer values and uses a millisecond precision, it was necessary to make small adjustments to the data structure. These include the previously mentioned block size increase to support milliseconds precision, but also when storing the values the notable difference to the algorithm is to the increase the value compression's leading zeros length encoding bit length which is in the original paper 5 bits and in our implementation it is 6 bits. This allows to store 63 as the leading zero value as the maximum, instead of the previous 31. That change also falls into the category of performance improvements. By increasing the leading zero values to 63 we can remove the requirement to do clamping like in the go-tsz and InfluxDB. This reduces one branch prediction from our write path. The original 5 bits works properly when storing only floating point values because the last mantissa bits are not often zeroes in the XOR result, but as noted in the original paper[92] many stored values are actually integers and it is more resource efficient to store them as integers instead of converting them to floating point values.

Original algorithm stores the timestamps by checking the range in which they fall such as [-64,64] and then further checks if the value is positive and in that case reduces the value by one to fit in the reserved bits. This means the algorithm does two branch predictions in the writing phase and in the reading phase it must do another branch prediction to check that if the value is positive then it must be

increased by one. In the implementation we have taken a different approach by first using ZigZag encoding[40] to always get a positive number and then considering the stored value as unsigned instead of signed. That value is then reduced by one to store it as [0,127] in the reserved 6 bits. In the decompression phase we increase the value by one and decode the ZigZag value back to the original value. This removes the need to do any branch predictions and gives us predictable performance, which is faster although we have to do some extra calculations as the CPU is able to do out-of-order processing.

Other notable differences in the performance come by processing the data a bit differently than other implementations. For example, in the timestamp compression phase we need to know how many bits are reserved for the value's range. The go-tsz[34] uses a simple branched execution until it finds a suitable value, Beringei[11] takes first an absolute value and then iterates over a table of possible choices to check if the value is lower than the maximum for that table value. In our implementation, we calculate the necessary bits by checking the amount of leading zeros and subtracting that from the length of int (32 bits) since we know the maximum length of a delta-of-delta value is at most 27 bits in a single encoded structure, otherwise it wouldn't fit to this block and would be encoded in the next block. This is optimized to native x86-instructions in the JVM and we can then do a simple lookup to a table to find the correct function without any need to do branches or iterations.

All the other implementations[11][34][96] operate using one-byte buffer which is then pushed to the underlying data structure after it has been filled. Instead, we exploit the idea from the Simple-family paper by Anh et al.[3] using machine length one word buffer as our storage and *snip-by-snip* encoding. This gives us performance benefit because we know all the values that are stored are at most 64 bits long and thus can be filled to the current buffer or the current and the next one. This removes the need of using loops to manage the byte-sequence and allows to use a single shift operation to arrange the encoded value to the buffer instead of multiple ones which span across bytes. While it is true that 64-byte alignments[136] can be even faster as buffers, we did not notice any notable speed improvements in doing so but it did complicate the code.

Original paper also notes excluded the use of more complex predictors to simplify the implementation. As seen in the tests on this paper, we noticed that the compression ratio of the first-order DFCM surpasses the last-value-predictor used in the original implementation. We did not find the first-order to be complex to implement, in fact it adds a negligible amount of code to the overall solution and the performance difference falls into the margin of error also. The downside is an increase in memory consumption if the prediction table must be kept in the memory for large amount of series. Due to this, we allow selection of the predictor when creating the compressor instance.

In Figure 8 we can see the effect of our changes compared to the original algorithm. The performance is increased by 167% using our version of the algorithm, with the same compression ratio in this dataset. Difference in decompression speed is slightly less, with an increase of 52% as shown in Figure 9.

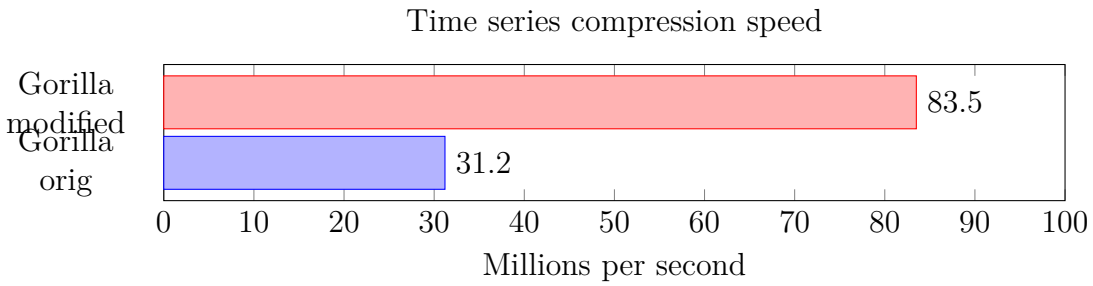


Figure 8: Gorilla time series compression speed

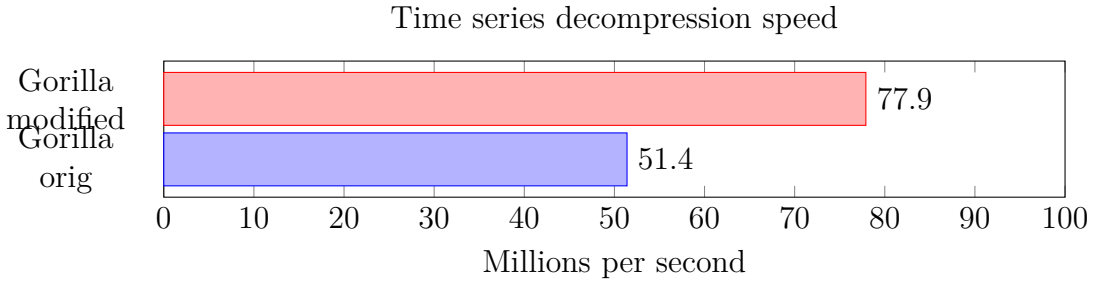


Figure 9: Gorilla time series decompression speed

5.1.3 FPC implementation

The FPC implementation is not used by our components as we discuss in the experiments and observations section. We built it to show results in this paper as the only available solution[73] for Java was ineffectively coded and would not have shown the algorithm in its best light. We rewrote the whole thing for performance, replacing multiple branch predictions with table lookups, precalculating certain values and allowing JIT to do effective loop unrolling and loop optimizations. We also improved the way bytes are written to the ByteBuffer stream for higher performance and removed unnecessary loops. The only interface change we did was to allow change of branch predictor table sizes as we noticed the original values were badly chosen for our use-cases and the original paper mentions other options for those values also.

We did not do any algorithmic changes so the implementation follows the original paper. Our implementation improves the performance for compression from 23.6M floating points per second to 55.6M per second, an increase of 135%. For the decompress we increased the performance from 25.1M to 61.0M per second, an increase of 143%. We have published our better implementation also to allow verification of the results in this paper.

5.2 Implementing compression to Hawkular-Metrics

Understanding the LSM behavior is important for understanding how Cassandra works and how to build stuff on top of it. We opted not to modify the Cassandra itself and instead build something on it to free us from maintaining Cassandra as well as our own solution. We followed the procedure of LSM by creating immutable

compressed blocks that would first gather information in sufficient amounts to allow proper compression ratio. This same approach is taken by some columnar databases, which first store data in a n-ary storage model (NSM) and later convert it to the decomposition storage model (DSM). For inserts, the NSM works well, since operations touch a single entity, like in our monitoring case a single update to a time series. However, for analytical queries and compression purposes, the DSM model is better.[10]

5.2.1 Data modeling

A traditional relational database data modeling that follows a conceptual modeling[23] is not a suitable strategy for Cassandra. The Cassandra data modeling is based on the idea of required queries to the data and to take into account the constraints of the Cassandra's query model, such as lack of joins or foreign keys. Efficient modeling requires the use of denormalization, data duplication and thinking about the structure of the SSTables.[22]

To represent uniqueness of a single time series, we use a composite key that is built from *metric name*, *data type* and *tenant id*. To fetch data from our tables, we require to know these three properties. Thus, we also provide multitenancy because all the data is partitioned by the *tenant id* for each time series. There is no restriction to have same *metric name* between different tenants. Since Cassandra is unable to cope with large partitions[21] we employ a pattern called bucketing[18]. Bucketing is a strategy that allows us to control the amount of data stored per partition and we use that as a part of our partition key, naming it as *dpart* and storing a bigint of calculated value (depending on the size of the partition).

For actual data in the cells, we store the compressed data in a byte-array column. This byte-array consist of a header part and the actual compressed data. In the header, we have reserved 4 bits for the compression method information and 4 bits for additional parameters. After that, the compressed data is present and that may include additional metadata related to the compression algorithm. 10

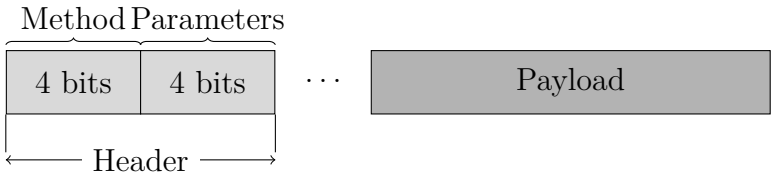


Figure 10: Compressed data cell

We support the data types of integers, floating points and enums (such as Availability) in the compressed data. If the selected compression format, such as our improved Gorilla compression, can compress both integers as well as floating points, the header parameter part has a bit flag set at *0x02*. All the enums are stored as integers using their ordinal number as the data. That means decompressing the enums back to their original format requires to know the original enum that was

stored. In our use, that is usually handled by the *data type* that we stored in the partition key. For timestamps, we process milliseconds since epoch.

5.2.2 Execution

When the server launches it schedules a repeating job that is ran every two hours at odd hours. In case the execution is missed at the scheduled time, the scheduler will immediately start the missing jobs in order with only one being active at a time (cluster wide, state is maintained in Cassandra). The job implementation itself will receive the original scheduled time of execution as its processing time and is not aware of missed execution window.

The job calculates the time window of previous two even hour window and calls a compression process to compress this block of time series. This allows at minimum one hour of delayed writes to reach the correct table before they are being compressed. For example, assuming the job starts at 03:00, we will compress the data points that were sent between 00:00 and 02:00.

5.2.3 Transforming data to wide-column format

To reduce the amount of storage used we needed to solve two things: we needed a way to reduce the amount of overhead per item and the amount of space required to store the data. To reduce the overhead we looked at the database industry which uses a columnar storage model to achieve this goal and decided to implement it on top of the Cassandra's LSM tree. A columnar storage format allows us to employ a more effective compression algorithms and to reduce the replicated metadata since each of our data point consists of timestamp, value and optional tags. With a default Cassandra 3.x row storage model[67] for each data point we store a row with cell values.

Each row requires a minimum of 6 bytes for row metadata and then each cell, such as timestamp and value, require one more byte each. In total, a minimum of eight bytes is stored as overhead for each row in our data model. It can be higher, such as with TTL, which we use also, but this is a good number to understand the issue. While this data is effectively reduced by the block compression, it is not continuous and as such requires at least a pointer for each row in the compressed format.

With Cassandra, it is not recommended to make a single row too large as it needs to be read in the memory when querying. We decided to store two hours of data to a row which made sense in the data patterns we are mostly seeing. This allowed us to get a large decrease in the overhead but still keep the size of a row enough small as to not directly cause issues with compaction or memory allocation. Larger blocks would have caused the rows to be larger but without significant benefit. For example, given 720 data points per block, we have now reduced the overhead per data point from 8 bytes (6 for row, 2 for cells) to 8 bytes per 720 data points, which is a reduction of 5752 bytes.

Since Cassandra is now unable to find the correct rows with normal queries, as it cannot look inside the columnar block, it was necessary to adjust both our querying

engine as well as the selected key for our storage. The row key was adjusted to be the start time of the two hour interval and all the time limited queries are rewritten to start of these row keys instead of the given timestamp. When the data is aligned this way, we will need a solution to align the data from different columns to match correct original row. For this purpose, we keep a counter of row position that is currently being scanned and reconstruct the data only if the given predicates for the scanning match. For example, our compressed row might have included timestamps that were not requested during the scan, but they do occur in each of the stored column sets. By keeping track of the original row position we know the original timestamp of each column and can then decide if we should include this column value in the end result. This logic adds some overhead in cases where we do not need to ignore certain parts of the data, but this was an acceptable trade-off.

5.3 Temporary tables solution

When a single table contains both, persisted as well as temporary data, we run into issues with compaction and tombstones. Due to the way deletes are handled in the Cassandra, removing temporary table actually requires another write to the table and that will be persisted until the tombstone timeout and full compaction of the block has happened. In most cases, this does not happen inside the TWCS window and as such we leave temporary data to the table until the TTL finally removes them.

During the time when there is both tombstones and live data, scanning for the keys and data requires Cassandra to access all possible SSTables to check if the partition key still has live data or if it is just tombstoned data, as well as polluting bloom filters with false positives. This increases the amount of I/O needed to run the compression job as well querying for the data from this table. In multiple systems, the simultaneous compaction and read load causes the I/O to spike beyond what the overprovisioned instances can provide and errors start to appear.

To solve this issue, we needed to find a solution that reduces the amount of I/O when doing the compression job as well as prevent read operations from accessing multiple tombstones. For this, we went for a solution that stores data to be compressed in temporary tables that can be dropped after processing and out-of-order data in a separate table.

5.3.1 Table management

Because the temporary tables solution requires that we always have an available table for writes, we opted for a solution that keeps temporary tables for the next 24 hours of writing. These tables are created by a separate job that runs every two hours and validates from the Cassandra cluster that there are enough available tables and creates the missing tables. This usually means creating one new table on each run, but a bootstrapping of a cluster will create 12 new tables.

At the same time, each Hawkular-Metrics instance is registering a callback to the Cassandra driver that monitors the schema changes. Once the listener notices that

tables have been added or deleted, the listener then proceeds to create new prepared statements for these new tables or deleting pointers to old ones that are targeting a removed table.

To maintain a set of prepared statements for each temporary table and to find a correct prepared statement for each query, we maintain a CompareAndSwap (CAS) variant of skip list[98] (using the Java's standard library ConcurrentSkipListMap[24]) that has as the key a timestamp of the block start and then as a value a HashMap[49] that has a unique identifier for each prepared statement. As the skip list is a sorted map it allows us to find a correct set of prepared statement by giving us the ability to look for a range of values. Because of the sorted nature of skip list we can always search for a preceding key or a range of keys without doing additional lookups.

Skip lists are an excellent data structure for our purpose. The interesting aspects of the skip list implementation are its lockless implementation possibilities with the use of CAS in the in-memory implementation as well as its sorted nature. They also require no extra processing to keep them sorted thus providing consistent speed when reading. As our skip lists are also small in size, the lack of good cache efficiency is no issue as the whole structure should fit in the modern CPU's cache lines.

The average complexity of operations such as finding the closest element to a given key is O(log N) and finding a range of keys is the same O(log N). Those operations are the two most common operations in our performance critical paths.

We extended the temporary tables with a fallback solution that will always be able to accept any out of order write to our tables. This is possible because we can write to a skip list with a key *0* that will always be matched as the last resort for the preceding key. In this case, if the data arrives after the temporary table has already been compressed to the final table we can still find a place to put the data to. That data can also be easily queried by looking at the correct key.

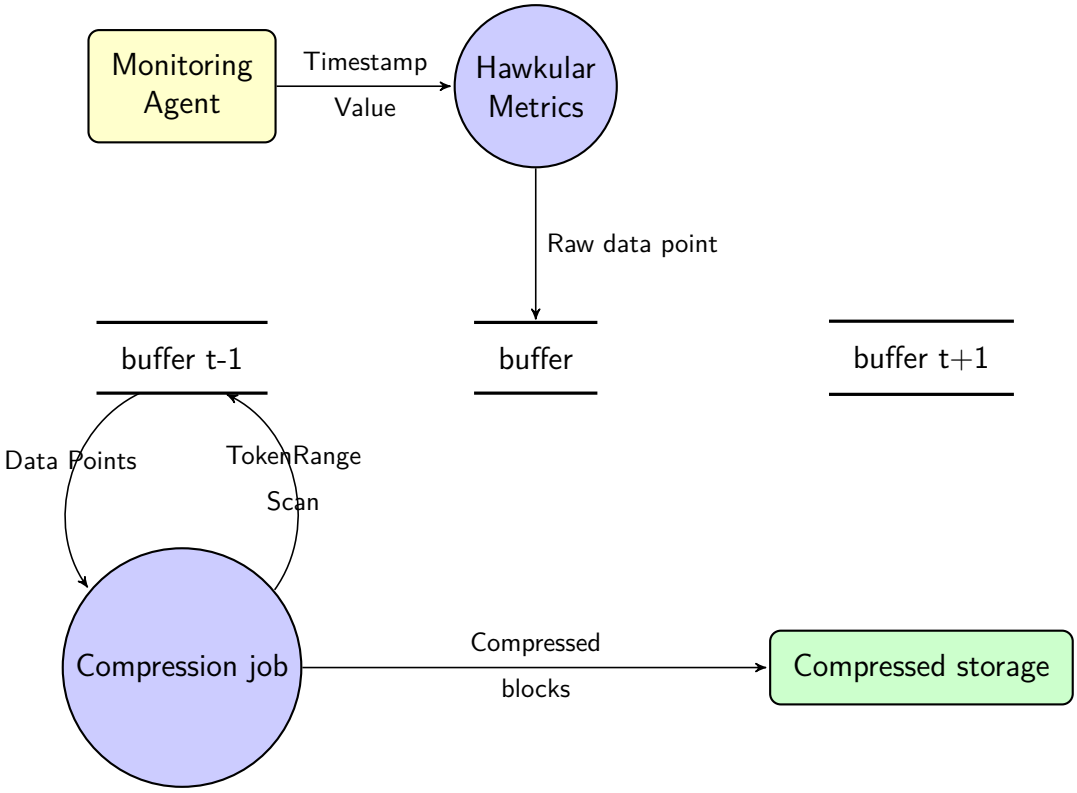
5.3.2 Writing data

When user writes a set of metrics to the database, each data point is capable of having a unique timestamp. Previously this has not meant much as it was just another bigint to store to the Cassandra, but the addition of temporary tables adds another step to the processing as we need to find out which table to push the write to. It is not necessarily the current timestamp table, but it could be the previous table or even the next timestamp range. This is especially true when we are near to the point of changing the temporary table in which case even a small difference in machine times can change the location of a write.

Thus, when writing each data point we must look at the skip list for a correct prepared statement which is used to write the data to the Cassandra and we do this by seeking an entry with the data point timestamp. This nearly never is an exact match which is the reason why we always seek the preceding key of the given timestamp. However, this also allows us to avoid calculating what the actual correct timestamp would have been as seeking the preceding key has the same complexity as seeking exact key. While it might have been possible to use a combination of exact calculation and a HashMap for near-constant seek time, that would have required

n-seeks to the HashMap when seeking for a range of keys which also would have had to be calculated and as such increasing the complexity of the implementation.

In case we do not find any temporary table there is a possibility of finding the preceding key with a key value of 0 which is reserved for our fallback case to the out-of-order writes table. This is user configurable, so if it is not found on the skip list we will ignore the data point.



5.3.3 Reading data

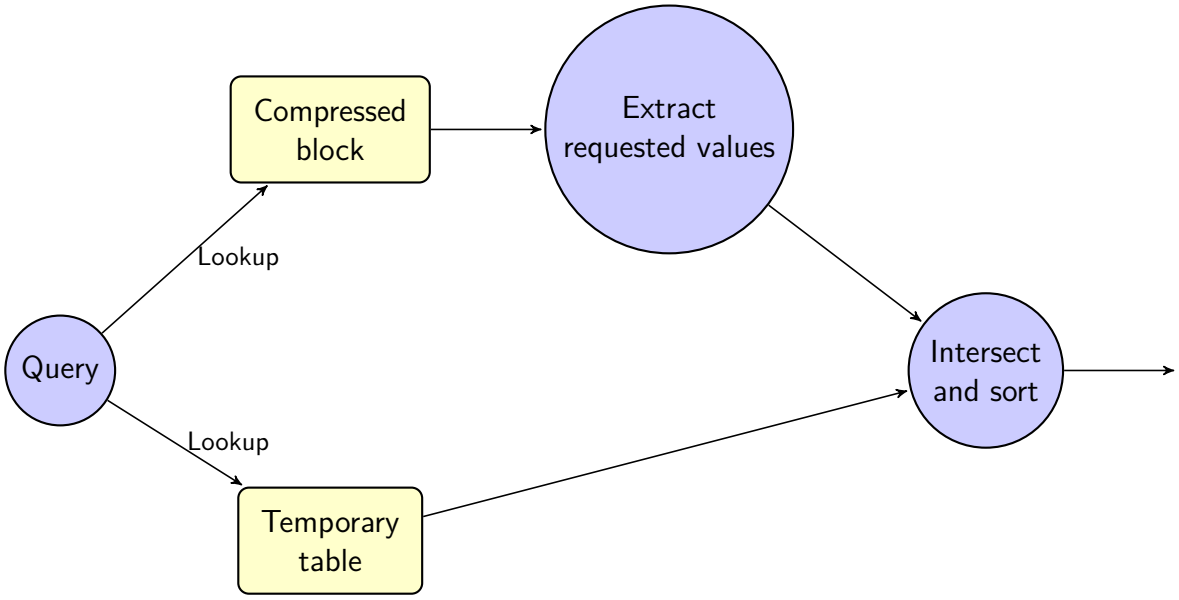
While the compressed data already presented us a problem of reading sorted data from multiple different sources, the temporary table further complicates this read path. We opted to create two different solutions for data reading, one for answering simple queries targetting a certain time series and one for streaming all the data from a table for external processes such as compressing the data to final destination.

When processing a request to read data between certain timestamps, we need to look at how the data is actually stored at the time of query. We look at the timestamp interval and then from the compression job results we first remove time ranges that the compression job has already processed and that are not part of the temporary table storage anymore.

Then, the remaining timerange is processed and we extract a list of all the temporary tables that are affected and split a single query to multiple queries that will affect different tables. While this is usually one or two temporary tables, we must take account the possibility of having delays in the compression job, in which

case we must process all the potential tables. We get this information from the job executor and when it last successfully processed the TempTableCompressor job. To account for delayed writes that were over one hour late, we also make a read request to the out-of-order table if such is configured to be used.

After selecting the correct tables, we will execute the queries asynchronously to all the temporary tables. Then, an equal process to the previously mentioned merging of asynchronous sorted streams is executed to get the data points in the requested order before merging them again with data with the compressed and out-of-order data writes. The sorting of a single stream of blocks is handled by the Cassandra's execution engine, as well as filtering inside a single temporary table, while filtering and sorting between tables and the compressed blocks is handled in Hawkular-Metrics.



5.3.4 Data partitioning strategies

Although it makes sense in the data compressed table to use multiple partitions for a single time series, in the temporary table we opted not to do that. Instead, we will keep all the data for a single time series together for more memory effective compression processing. While this creates a possibility of hotspotting a single node in the cluster if one time series is taking a large amount of total processing power, we deemed the possibility to be quite small for a single series to cause issues.

Instead, with a single time series in a single partition we can query all the data of a single series by using token ranges when doing a compression job. This allows us to get all the data points for one series in a sorted order and then the next time series and so on until the whole token range has been processed. This then gives us the possibility of asynchronously combining all the data points directly to the compressed format and then discard the data point and after the time series ends, we can discard all the pointers to the compressed container also. This gives us a

huge boost in the memory effectiveness as the heap can now be cleaned by the GC constantly without needing to promote any objects to old gen.

The reduced GC pressure allows us to compress the data quicker and with smaller amount of memory, thus increasing the stability of the application also by not inducing extra memory pressure on top of the normal processing. Smaller GC pressure also frees CPU cycles to process the actual data and reduce the overhead of a monitoring solution thus giving the customers the ability to use that freed memory and CPU resource for their primary workloads.

To verify our hypothesis of reduced GC pressure and that it fits our reactive processing methods, we setup a test to write 200 000 new metric time series every 10 seconds and let it run for 36 hours. For our older method, we used a 4GB of heap while with the temporary tables solution we only allowed the system to use 512MB of heap. During the 36 hours of processing we generated 144 million new time series to each table and a total of 2.5 billion new time series. Our older method choked eventually after 6 hours of processing and spent most of the CPU time handling internal bookkeeping objects. No such issues were seen with the newer method, where the processing had no increase in latency over the 36 hours of processing and the processing did not fail at any point.

6 Experiments

For readability, certain figures are not zero based. The figures use different units depending on the test. In compression ratio tests, lower results are better while in performance tests, higher results are better.

6.1 Data set compression ratio

We look at the compression ratio separately for timestamps and values. All the values are in bits per type, meaning we report how many bits it took to compress a timestamp and how many bits it took to compress a single value. Those values are only applicable to the block size that was measured as each compression algorithm has their own set of metadata overhead that plays a different significance depending on the block size.

Timestamps are compressed by first calculating the delta-of-delta of values between the timestamps and ZigZag encoding them to make negative values positive and then compressing the resulting values. Although results are omitted from this paper, this significantly reduces the amount of bits required to store the timestamps[92]. It does not affect the compression ratio comparisons as all the algorithms benefit from this, thus comparing the actual encoding format only.

6.1.1 Generated data sets

All the generated data sets are measured and compared without the overhead of Cassandra, thus only the real payload is measured. Also, 64 bits for the original timestamp and a 32 bit word for the first delta are added to the values of Simple-8b and PFOR, since they are both word or aligned codecs in their implementations. Gorilla compression uses 64 + 27 bits for the same information. Generated data sets are run with different block sizes, 180, 360, 720, 1440 and 2880 data points. This equals 10 second measurements in 1 hour, 2 hour, 4 hour and 8 hour blocks.

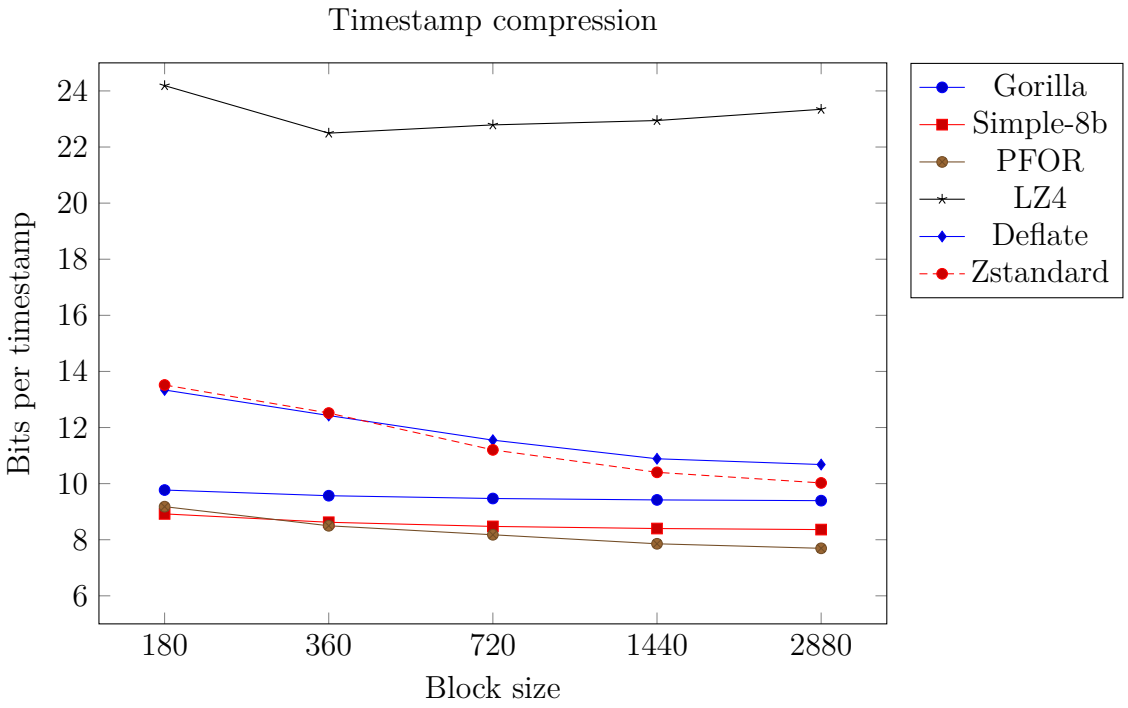
Zipf sample. The data set's values are generated by using Zipf distribution as mentioned in the synthetic data sets section and to make it slightly less compressible, the data contains 100 000 unique values and a skew of 1.9. For timestamps, the delta between values is always $60 * 60 * 1000/S_{seconds}$ milliseconds, but adjusted with a randomized value of +-25ms for each value, with $S_{seconds}$ meaning the size of block in amount of data points. Due to the slightly changing random nature of the data, all the tests are run 10000 times, but equal data is always used to measure single round of execution. After the each run, the amount of bits used by the compression algorithm is recorded using HdrHistogram with a precision of 3 digits. The amount of unique values per block can be seen from the Figure 11.

Removed from the statistics is the Simple-8bRLE variant, which was also tested but it returned the exact same results for this dataset as the default Simple-8b. This indicates there are no long runs of any unique value in the sequences. From the results we can see that each compression algorithm beats the Shannon Entropy and larger block size has an improved effect on the compression ratio Gorilla's encoding

Block size	Mean Unique values
180 data points	99.9
360 data points	131.9
720 data points	153.1
1440 data points	167.0
2880 data points	176.6

Figure 11: Statistics of Zipf generated data sets, mean values

Figure 12: Zipf timestamp compression results



method has the worst compression ratio in all of these tests, ranging from 10.7% difference to winner in 180 data points blocks to 18.1% difference in 2880 data points test. Simple-8b is the best compression algorithm in 180 data points test with 2.8% margin while losing as much as 8.0% in 2880 data points test to PFOR. PFOR provides the best compression ratio from 720 data points forward, while 360 data points results can be called even with Simple-8b since the results fall inside the standard deviation.

The general purpose compression algorithms cannot quite match the specialized compressors in terms of compression ratio. They show however slightly larger benefits from increased block size. Out of interest we even tested up to 46080 block size to see if Zstd would reach the compression ratio of specialized compressors, but the scaling reaches a plateau and will not reach the specialized compressors.

InfluxData comparison. The creation process for this data set is described earlier in the data sets section. Due to the low number of integer values in the set, we opted to use this data set as an example of storing everything as floating point. This is common behavior for many collection agents as they might not know the original type of the data and thus revert to the floating point for safety reasons. Results are visualized in Figure 13.

We did run the test for multiple block sizes, but omitted them from here due to relevancy. The block size had no effect on the order of the compressors or their relative difference. Between the specialized floating point compressors, we see an advantage of 41% for the Gorilla when using DFCM predictor compared to the FPC which uses the best result between FCM and DFCM predictors. FPC is no better than LZ4 in this data set and worse than Zstd or Deflate, which are both generic compressors. Zstd as the best generic compressor in its slowest and highest compression setting is still 12% to 20% behind Gorilla implementation.

Although excluded from the graph in Figure 13 we also tested the effect of cascading compression when used together with Gorilla and FPC. When FPC's output was compressed by Deflate, we saw an increase of 44.0%, but the Gorilla output was only reduced by 8.7%. With LZ4, we saw 27.4% and 3.5% decrease. Even with the cascading compression, the FPC is not able to produce better numbers than Gorilla with the same cascade combination. Actually, FPC with Deflate is not able to bypass Gorilla+LZ4 combination when it comes to the compression ratio.

6.1.2 Real world data sets

Openshift failed run. The data in figures 16, 15 and 14 provides a very different workload and such, results that do not necessarily follow previous runs. Here we see that the high number of repetitions while containing large numbers every now and then is something the specific compression methods are unable to handle well.

When compressing the floating point data in Figure 14 all the compressors are quite even in this benchmark, with the exception of FPC which is unable to compete with the rest. The difference between the worst (Gorilla DFCM) and the best (Deflate) is only 26.5% decrease in total bits used.

In timestamp compression seen in Figure 15 the generic compression methods

Figure 13: InfluxData comparison set value compression

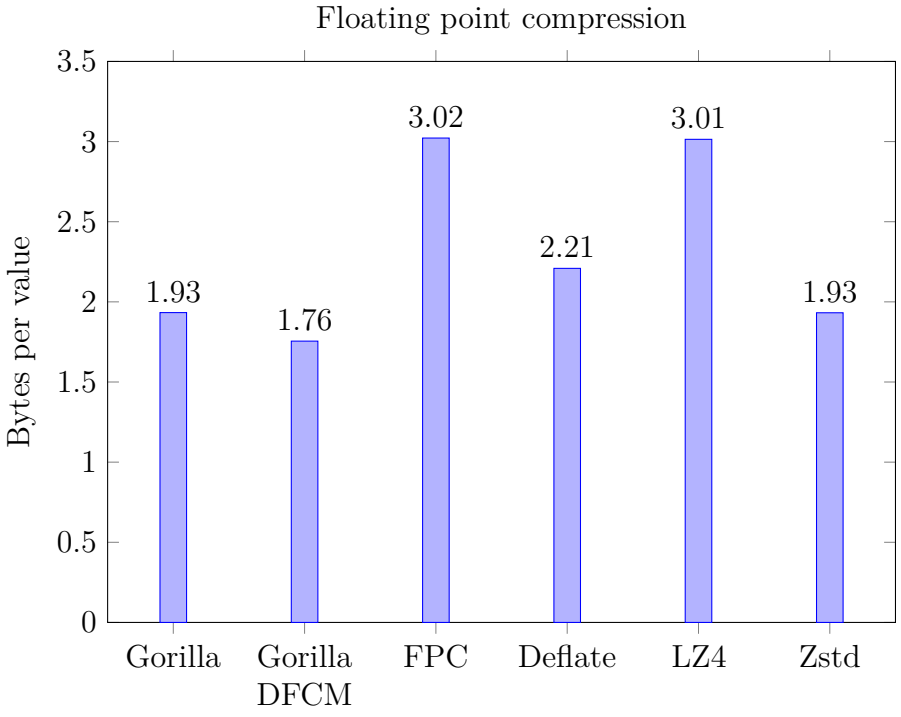


Figure 14: Openshift failed set floating point compression

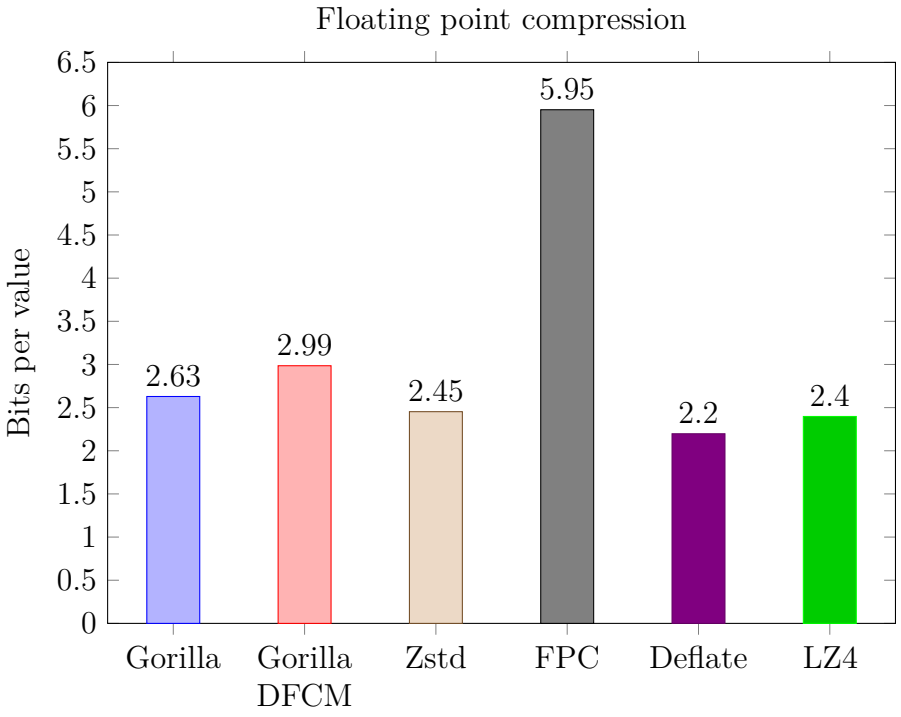


Figure 15: Openshift failed set timestamp compression

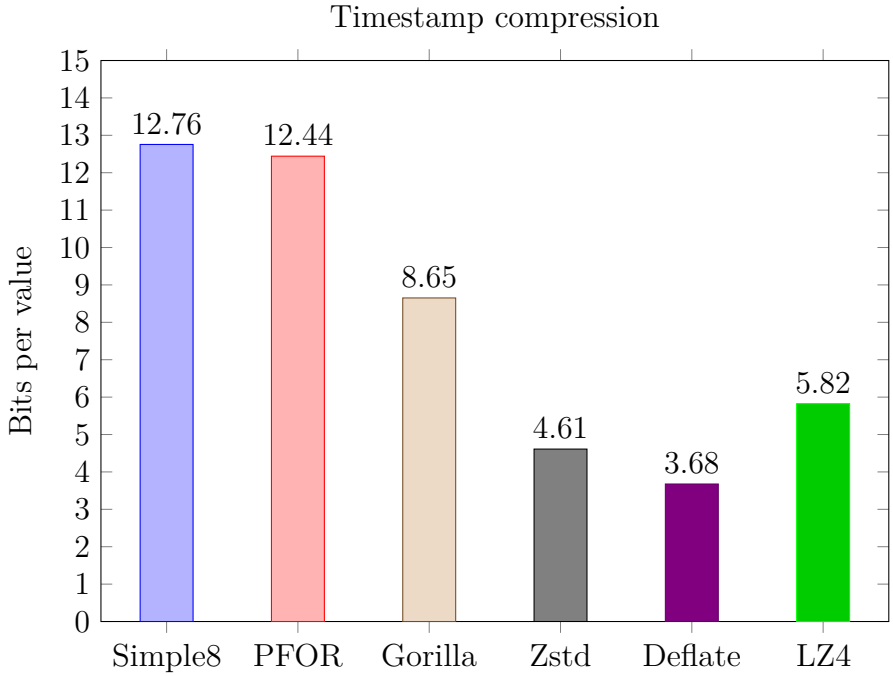
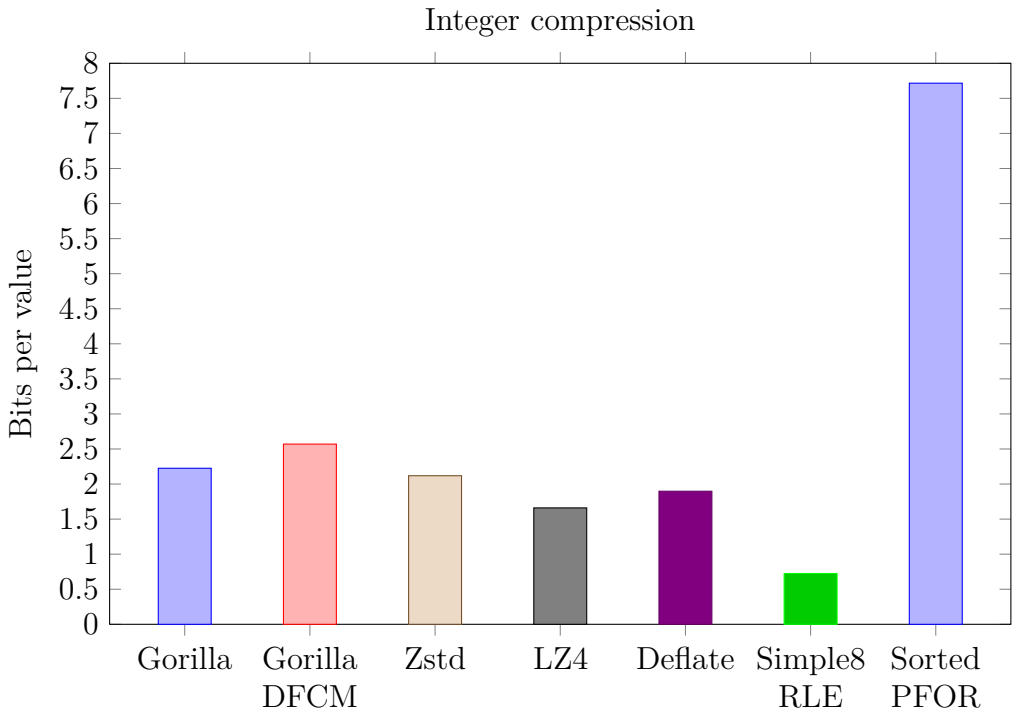


Figure 16: Openshift failed set integer compression



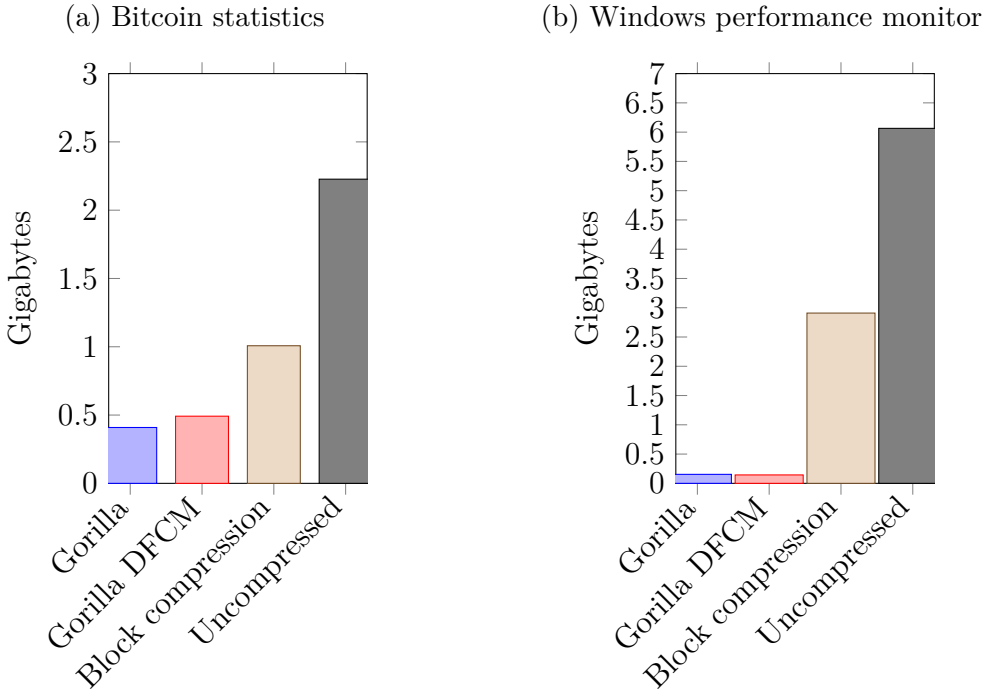


Figure 17: SSTable sizes, lower is better

are superior to the specific methods, but only if we reduced the entropy of the values for generic methods also. Thus, all the values have delta-of-delta applied to them before compression. Without the delta-of-delta, the Deflate compressed with the data set using 30.3 bits per timestamp. Thus, we see almost a magnitude of difference when using delta-of-delta encoding for Deflate, which is the best compressor here. Gorilla timestamps compression has delta-of-delta built in, so we did not apply preprocessing to its dataset. However, after that processing, the Deflate still takes only 42% of the total space compared to the results from Gorilla, which is the best specific compressor.

From Figure 16 we have excluded PFor with mean of 33.3 bits and Simple-8b with 50.1 bits for readability reasons. As we can see, the addition of RLE algorithm to the Simple-8b creates a very large difference in the overall compression efficiency as the blocks align much better. The sorted PFOR has delta coding enabled, which provides improved results over normal PFOR indicating the integer data has growth patterns in it. The best compressors are LZ4 and Simple-8 RLE, both which have special processing for repetitions.

In these setups, we saw that in both, integer compression 16 and in floating point compression 14 the last-value-predictor is better at predicting the next value than more advanced predictors such as DFCM.

Bitcoin values. All the results in Figure 17a are reported as resulting SSTable sizes in the Cassandra storage. This includes all the overhead metadata, and all the files have been compressed with LZ4, excluding the Uncompressed results. The LZ4 reduces the file size for native approach by to 45% of the original size, but only

to 93% for Gorilla results. We see that previous value is the better indicator than DFCM for this dataset. In Gorilla results, around 18.5MB of the total results are spent in the Cassandra overhead, rest are actual payload when Gorilla compressed. This allows to compare the results to other solution. On average, 0.24 bytes per data point (including timestamp and value) is spent on Cassandra overhead. Also, on this dataset average amount of space per data point for Gorilla is 5.46 bytes before block compression and for DFCM variant it is 6.55 bytes. Thus, we reduced the original file size to 18.3% from uncompressed size, but only 40.5% of the block compressed version.

Windows performance monitor. All the results in the Figure 17b are reported as resulting SSTable sizes in the Cassandra storage. This includes all the overhead metadata, and all the files have been compressed with LZ4, excluding the Uncompressed results. Here we see that LZ4 compression reduces the uncompressed blocks to 47.9% of original size. Gorilla compresses to 2.6% of the original size or 5.4% of the block compressed version. In this data set, DFCM version does slightly better, compressing to 2.4% and 5.0% respectively. Both Gorilla compressed results benefit from the LZ4 more than in Bitcoin results, with a reduction of around 33%. Before LZ4 compression, 54MB is spent on Cassandra overhead. Gorilla spends 1.46 bytes per data point while DFCM versions uses 1.36 bytes per data point.

6.2 Performance of the compression job

We selected two datasets to compare the compression speed. For floating points we used the InfluxData comparison data set, while for the integers and timestamps we used the generated Zipf dataset. We chose not to use any real-world data set for these benchmarks as these are theoretical in nature also and lack any other overhead coming from the processing of the compressed / decompressed data in the pipeline. The results are presented in how many items we are able to process per second (in millions) instead of how many bytes we are able to process per second as we process everything as numbers. Thus, the compression ratio might have an effect on the performance results.

InfluxData comparison. Each of the block in the data set consists of 720 data points. The compression ratio can be seen in Figure 13.

From the results in Figure 18 we can see that in the generic compression algorithms, there is a large difference among the compressors. Zstandard is 7.5 times faster than Deflate while achieving better compression ratio as seen in the previous chapter and LZ4 provides a magnitude of better compression speed compared to the Zstandard. LZ4 is even able to bypass the FPC in speed, while achieving equal compression ratio. In this group, Gorilla achieves the best compression speed, besting LZ4 by 18,5%. This applies to both versions, with DFCM as well as with Last-Value predictor, between which we could not measure meaningful differences in speed and as such have published only a single result. They both achieve the best compression ratio in this test as seen from Figure 13 as well as the best compression speed.

In Figure 19 we notice that the slow compression speed does not necessarily equal slow decompression speed. Generic compression methods are all providing quite

Figure 18: Floating point compression speed, 720 data points per block

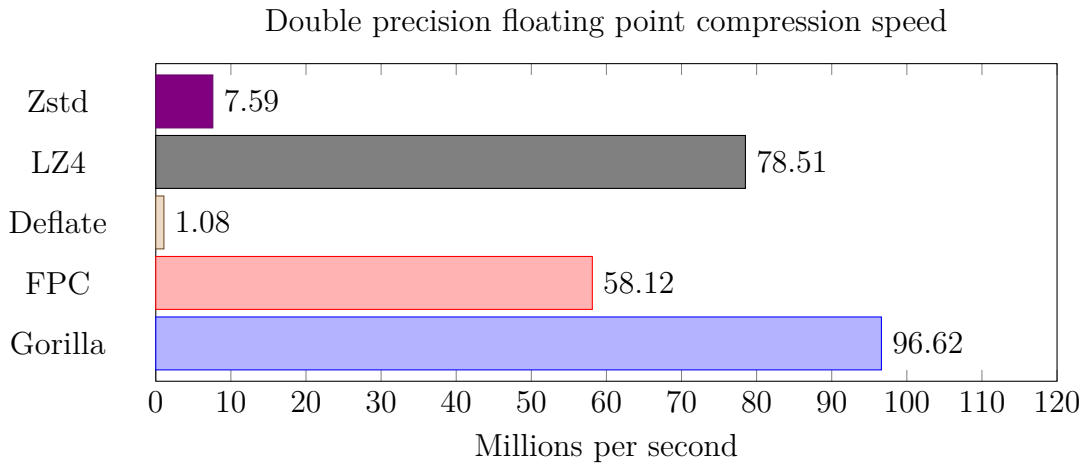
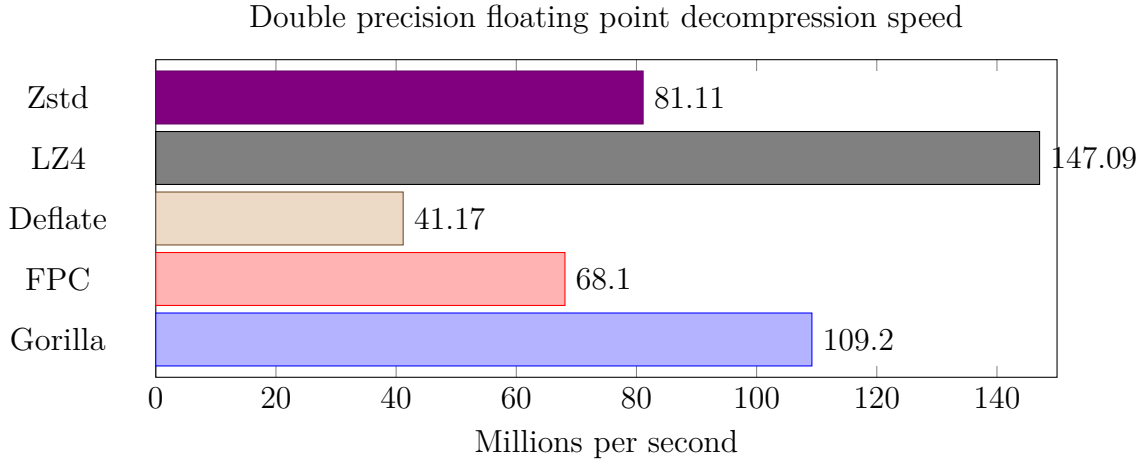


Figure 19: Floating point decompression speed, 720 data points per block

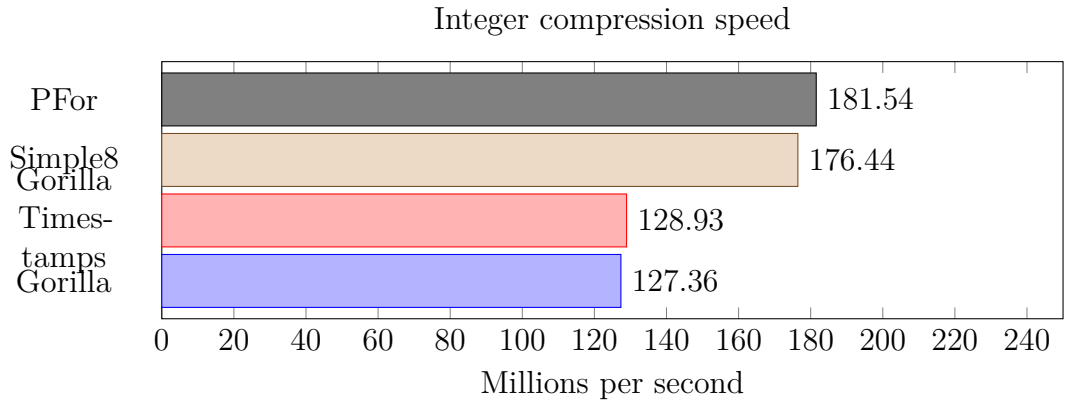


good decompression speed, with LZ4 actually being the fastest option here - even compared to the Gorilla. Zstandard is also able to provide good speed and bypassing the FPC for performance. This is partly explained by the bad compression ratio of FPC as seen in Figure 13 which causes the FPC to process 63% more data than what Zstandard needs to. We saw no difference in DFCM or Last-Value-Predictor decompression speed when it comes to the Gorilla, and although DFCM requires slightly more processing it also compresses slightly better so there is less data to process. LZ4 is 81% faster than Zstandard and 34.9% faster than Gorilla.

Zipf samples. Each of the block in the data set consists of 720 data points. The compression ratio can be seen in the Figure 12.

When it comes to the integer compression speed from Figure 20, we see quite even results between different compression methods. The results for generic compression methods as described in Figure 18 as they did not differ compared to the integer results. The results for Gorilla timestamps are not comparable to other results as it has delta-of-delta already calculated. The comparable results for Simple-8b and

Figure 20: Integer compression speed, 720 data points per block



PFor would be 130M and 132M timestamps / second respectively, with LZ4 clocking at 55M timestamps / second as the fastest generic compression method.

In value compression results, we see PFor leading the pack, but the Simple-8b is only 2.8% slower than PFor. Gorilla is slightly behind, with 29.8% slower performance compared to the PFor. LZ4 even as the fastest generic compression method is still 56.9% slower than the PFor. For timestamps compression results, there is only a small difference between different compression methods, with PFor still being fastest, but Simple-8b is only 1.5% slower and Gorilla lags behind only 3.0% compared to PFor. LZ4 is 58% slower than the PFor and rest of the generic compression methods are far slower.

Figure 21: Integer decompression speed, 720 data points per block

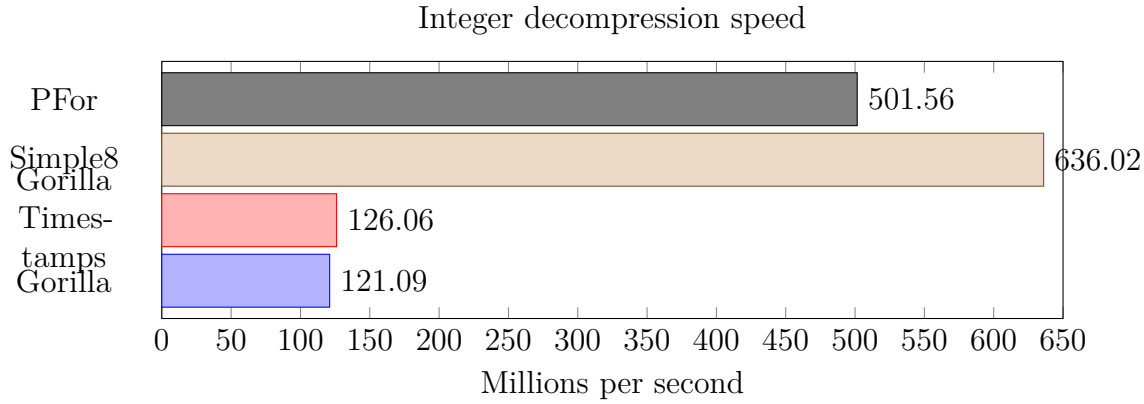


Figure 21 shows the performance in decompression of integer compressed data. We have omitted the generic compression methods from these results as their speed was equal to the ones listed in the floating point compression results in Figure 19. Gorilla is listed twice, for integer value compression and for timestamps separately, since depending on the stored data a different method is used. Like in the compression results in Figure 20 the Gorilla timestamps results are not directly comparable to the PFor and Simple-8b results as their compression would require an extra step of

prefix calculation twice to regain values from delta-of-delta compression. This would reduce their performance to 355M timestamps / second for Simple-8b and 308M timestamps / second for PFor, with LZ4 dropping to 124M timestamps / second.

From these results we can see that the value decompression is fastest with Simple-8b, which is 27% faster than the PFor. Both are quite a bit faster than Gorilla, Simple-8b by almost 5.3 times. For generic compression methods, the results from Figure 19 show that only LZ4 is comparable in performance to the Gorilla results, but loses to specific integer compression methods. When comparing the results of timestamps compression, the added time for calculating prefix sums from delta-of-delta reduces the advantage of Simple-8b and PFor compared to Gorilla. Simple-8b is still 2.8 times faster than Gorilla's timestamp compression method and 15% faster than PFor. LZ4 and Gorilla are even when it comes to the compression speed with timestamps.

6.3 Query performance

There were some fears in the development team regarding the effect of the compression to the query performance. Some queries only read partial amount of what a compressed column would store, while some of the queries might be analytical type and read large amounts of data. As a baseline, we will use the uncompressed data in the Cassandra, stored as normal rows. The data set is generated by using Zipf distribution as mentioned in the synthetic data sets section and to make it slightly less compressible, the data contains 100 000 unique values, 360 data points per hour and a skew of 0.9. Using a smaller amount of unique values and a larger skew would have reduced the compression size of the dataset and thus further benefit the compressed queries. As an example, reducing the unique values to 1000 and a skew of 1.2 reduced the data size by approximately 31% while it reduced the uncompressed size by only 12%. For timestamps, the delta between values is always 10 seconds, but adjusted with a randomized value of +-50ms for each value. This tries to mimic the behavior of a millisecond precision storage for monitoring polling intervals.

In this experiment, the only calculation done to the data is a sum $\sum_n^0 x_j$ which is a simple calculation and thus ensures we measure the Cassandra data fetching and decompression job instead of the actual calculation. The experiments were conducted in a single machine configuration, thus reducing the effect of the network communication also to provide more stable results. The pipeline of the test also removes the REST-layer processing of Hawkular-Metrics, only concentrating on the actual backend query processing. Although the results here are provided only for the Gorilla compressed blocks, but the results can be transferred to other compression methods also using the previously known data from compression ratio and performance as they were measured using full blocks of data. The overhead for Cassandra does not change with the used compression method and is always compressed in these examples as LZ4 is used as a block compression in all of these examples, including the uncompressed results.

All the queries are warmed up by executing them first 1000 times and then sleeping for 2 seconds to allow GC and JIT some time to stabilize. After that, the

calculations are done 10 000 times and results are stored in a histogram by measuring the latencies using nano seconds from the OS. Using the nanotime measurements gives us around 15ns granularity on the JVM running in Linux. [85] The histogram implementation we used to record these values is called HdrHistogram[53] and we defined a precision of 3 digits to it during our testing. That is, our value quantization is no larger than 0,1% of any value. For these reasons, the timings in the table are given in milliseconds with three digits.

Looking at the actual data sets in Figure 22 , we can see that the compression ratio benefits are diminishing quite quickly after the first hour and the size of the data is not a significant modifier in the experiment. More interesting is the amount of partitions, which directly translates to the amount of queries executed to the Cassandra. For each compressed data set, a single query returns a single row from Cassandra with a single column and for the uncompressed, 360 rows are returned per query with a single column.

Hot data. Before the data was processed, all the data was first flushed to the disk and compacted to a single SSTable per measured compression block size. All the reads were from a page cache of the operating system, thus the disk subsystem's speed made no impact to the results. Data was partitioned to a single compressed block per partition and one (1) hour partition for the uncompressed data.

The figures 23, 24, 25, 26, 27 show five tables showing the execution time to read and uncompress the data and calculate the results. When comparing the results of uncompressed data and one hour compressed blocks, we see that there is a reduction of 87% to 92% in execution times. The one hour block is an interesting comparison as it executes the same amount of queries to the Cassandra and has the data partitioned to the exactly same amount of partitions. With larger block sizes we see a diminished reduction in the query performance compared to the baseline (uncompressed), however compared to smaller blocks we still see noticeable difference. For example, in the query for 180 days of data, the baseline difference of one hour is 91.6% and 95.6% for the eight (8) hours blocks. In other words, the eight hour blocks' query time has been reduced by 47.6% from the one hour block query time, but only 12% from the four hour block size.

In the Figure 28 we can see the percentage of time spent in decompression in the 180 days query. As seen in the query performance results, the less rows we fetch the less time is spent in total. The amount of data points to be decompressed is not reduced however and that means the decompression job takes a larger and larger portion of the total time in processing.

Figure 22: Generated data sets for query performance experiment

Block size	Size of data (bytes)	Partition count	Row count
Uncompressed	22580399	4320	1555200
1 hour	6176309	4320	4320
2 hours	5980608	2160	2160
4 hours	5870896	1080	1080
8 hours	5806372	540	540

Figure 23: Reading one hour of data, all times in milliseconds

Block size	Min	Max	Mean	99% percentile	σ
Uncompressed	0.172	14.737	0.775	1.592	0.425
1 hour	0.160	15.114	0.765	1.33	0.336
2 hours	0.00663	0.219	0.0384	0.0741	0.0196
4 hours	0.00702	2.822	0.0387	0.0740	0.0335
8 hours	0.00639	3.278	0.0359	0.0704	0.0403

Figure 24: Reading 24 hours of data, all times in milliseconds

Block size	Min	Max	Mean	99% percentile	σ
Uncompressed	5.456	24.986	9.989	18.924	3.352
1 hour	0.660	16.179	1.304	4.846	0.734
2 hours	0.535	10.830	1.074	3.121	0.486
4 hours	0.416	11.198	0.939	2.204	0.430
8 hours	0.292	11.469	0.813	1.898	0.414

Figure 25: Reading 7 days of data, all times in milliseconds

Block size	Min	Max	Mean	99% percentile	σ
Uncompressed	50.627	101.319	69.170	90.767	7.224
1 hour	4.297	21.561	6.888	15.565	2.082
2 hours	2.728	15.466	4.780	11.100	1.346
4 hours	2.367	14.762	4.004	10.576	1.252
8 hours	2.294	14.377	3.456	7.365	1.041

Figure 26: Reading 30 days of data, all times in milliseconds

Block size	Min	Max	Mean	99% percentile	σ
Uncompressed	277.086	414.712	322.929	381.420	19.550
1 hour	18.629	44.007	27.815	39.682	3.942
2 hours	14.524	37.257	20.465	30.540	3.041
4 hours	10.052	28.099	16.585	25.657	2.456
8 hours	9.708	29.524	14.510	22.823	2.238

Figure 27: Reading 180 days of data, all times in milliseconds

Block size	Min	Max	Mean	99% percentile	σ
Uncompressed	1774.191	2262.827	1896.954	2023.752	51.304
1 hour	111.084	206.307	159.003	182.845	9.373
2 hours	81.002	152.044	114.023	131.531	7.737
4 hours	72.810	302.514	94.714	107.938	8.638
8 hours	69.403	108.724	83.354	95.814	4.766

Figure 28: Percentage of processing time spent in decompression

Block size	% of time
1 hour	56.8
2 hours	63.1
4 hours	70.1
8 hours	76.8

7 Observations

Type of data affects the compression ratio provided by different algorithms and as such these results are not indicative of any other workload. There are large differences when it comes to the compression ratio between different data sets and as such any solution is always a compromise of certain features.

7.1 Compression ratio

In the InfluxData data set we saw somewhat interesting results when it came to the compression of floating points (Figure 13). We were negatively surprised by the bad performance of the FPC, which is in many papers compared as one of the best algorithms for lossless floating point compression. We speculate that the encoding format, which only suppresses the leading zeros from the compression results is the key here. It appears the data set values are predicted enough well for the trailing zeros to account large amount of the stored space and the Gorilla's algorithm is much more efficient because of this. The other reason is that Gorilla can store correctly predicted value with less bits and this happens if the value is repeated often, as is the case in monitoring data. Use of larger prediction tables made no difference in either Gorilla's or FPC's case, since the blocks are not large enough to fill a larger prediction table. The FPC required the use of cascaded compression methods to get close to the Gorilla's compression ratio, but never surpassing it while requiring more time to process. Also, the use of gFPC is not possible due to small block size.

The Openshift data set provides results that give more complex predictors and encoders found in generic compression algorithms a better chance in compressing the data. Since the blocks of data are smaller than in runs that are full of data, the results emphasize small block overhead as well as good prediction for RLE. Most integer compressors are not able to compress this type of data too well, with the exception of Simple-8b RLE variant, which can take very long runs of data and compress that to a single 64-bit word. Our results show that the combination of RLE (or LZ77 based solution) on top of these integer compressors could give us very good results in this benchmark, as it removes the largest disadvantage that they have compared to the generic compressors.

We also saw that in the Openshift runs the DFCM was worse option than the Last-Value-Predictor, while this was not true for the InfluxData comparison set. Thus, the choice between these two predictors for Gorilla is not as simple. The difference in either case is not remarkable however. We also tested the combination of Gorilla+LZ4, in this order, to see if we would gain the best of both worlds. In cases where LZ4 was better than Gorilla, we saw the Gorilla results only drop into the same level as LZ4. However, that means the combination has relatively few weaknesses also even if it can't provide always the best results.

For data sets outside the monitoring range, we see that bitcoin data is much more difficult to compress for these algorithms. It also highlights the problems of DFCM when size of the block is small, in this case only 120 data points per block. In those cases, DFCM is unable to fill the prediction table effectively to gain good

prediction performance. This is in contrast to the Windows performance monitor data set, where the resulting file sizes are just few percentages of the original data size, giving us very good performance.

A very important note to improving our storage performance is to look at the Cassandra overhead reduction. For example, in the Windows performance monitor (Figure 17b) data set, each data point uses 29 bytes on average for the storage. In other words, 16 bytes are used for the value and timestamp, while the rest 13 bytes are used for primary keys, sorting keys, TTLs, row encoding and such. Using only our own data presentation model, we reduced this amount from 13 bytes to 0.27 bytes per data point and in the Bitcoin statistics (Figure 17a) we used only 0.24 bytes per data point as overhead. That itself is a major contributor to the reduced disk space. We can also see that LZ4 is capable as block compressor to reduce this same overhead, by decreasing the amount of bytes used to 14 bytes from the original 29 bytes.

7.2 Compression speed

When compressing very small blocks, the overhead of many compressor implementations pay a large role. For example, reducing the predictor table size in FPC from 64kB to 8kB improved the performance almost ten-fold. This is because allocating 64kB of memory for each small block takes a lot of time and the data sets are too small to take advantage of the larger predictor table sizes. Using even smaller prediction tables made no noticeable difference in performance. Due to small blocks we see that the overhead plays a crucial role. Since we used a real world setup where we can not do inplace updates and reuse the same arrays, we see a performance that is much lower than what the authors of the respective algorithms might advertise. This applies to our own Gorilla implementation also, which can with larger blocks achieve somewhere around 1.3GB/s performance on compression on the same hardware when using different data set, but only slightly under 1GB/s on these tests. FPC's performance could be improved by the use of pFPC[16], a slightly modified parallel implementation of FPC algorithm. This does not apply to our use cases however, as adding threading to very small computations generates more overhead which takes away all the performance benefits.

Similarly, the generic compression methods suffer from their generic nature and complex algorithms. Only LZ4 can process speeds that are in the somewhat same league as integer/floating point specific compression methods, but in most cases the trade-off is then large loss in the compression ratio. And although Zstandard is designed for new modern CPUs and in this case was run using native-processing (and not Java) it was easily a magnitude slower than the specific compressors. This limits their benefit as we lose the great advantage in query performance and also we see increased CPU usage in a environment where monitoring for some occasions is seen as only a necessary devil and nothing more. However, since LZ4 shows some interesting behavior in the cases where our specific methods are unable to compete and with a suitable compression speed it is possible to run a dual compression algorithm with its performance. Since Cassandra supports compressing metadata and payload with

LZ4 in blocks, it appears to be a good compromise for many cases. Especially since we can control the block size freely - thus finding a good compromise between read performance and compression ratio.

Zstd is an interesting case as it can often reach close to same compression ratio than the maximum deflate, but at the same time it is a magnitude faster when using the level 6 in compression. In decompression the difference is slightly smaller, but still noticeable. We did not use multithreaded version of Zstd in our tests as the benefit compared to overhead does not do anything good for us. For larger blocks, such as compressing the Cassandra's SSTables we believe the multithreaded solution could be very useful. Whether it is useful enough compared to LZ4 in that scenario is something that we did not investigate as part of this project, however we did compare LZ4 and Deflate and found around 15% gain in the temporary tables. The loss of performance was too much for our interest at that point, but the Zstd should offset this somewhat. Although, many users are struggling to keep up with compaction jobs and Zstd will still slow down the process compared to the LZ4.

Integer compression methods are often very simple in nature, yet they show quite an effective compression ratios despite of that. Due to their simple nature, they are often easy to optimize to the newest processors and they are not often depending on the previous written data and as such can use out-of-order processing as well as vectorization. If performance is the key thing, these compressors are hard to bypass as they provide performance that far exceeds any I/O device where the data is currently stored in. Sadly, due to the nature of floating points the same does not apply to the floating point compressors where the only main ways of getting better compression ratio is to calculate differences from previous values and predict the outcomes. This means that the out-of-order execution is not often possible, branch prediction mistakes happen and no vectorization can be used as it is impossible to process multiple data points at the same time. As such, these specific compressors are not that much faster than generic compression methods and often behave exactly like them.

7.3 Selecting the compression algorithm

When selecting the compression algorithm, we give more weight to the performance with smaller blocks as that seems to be what our users are currently targeting. Most of the specific methods provide a suitable compression performance and their performance difference might be lost when combined with the overhead of the Cassandra. None of the generic compression methods however provide us with enough lightweight approach as each of them would severely reduce the speed of the compression job, which could leave us in a situation where we ingest more new data than we can compress.

The Gorilla compression has the disadvantage in being a streaming compression algorithm when it comes to the compression ratio. It cannot use the knowledge from more than one previous value to calculate the beneficial encoding sequence for the next value. This is clearly visible in the compression ratio tests, where it can fall behind the other compression algorithms when it comes to the timestamp

compression ratios, as it cannot benefit from the reduced amount of unique values when compressing timestamps. Streaming has also implications to its encoding and decoding speed, since it is a variable bit-aligned codec. This means we have to read a single block of data, be that metadata or value before being able to process to the next one. This is in contrast to some fixed-size encoding methods where it is possible to pre-calculate the following positions without reading any data. The benefit on the other hand is that it uses less memory since we can remove the uncompressed data from the memory instantly.

Streaming nature of the Gorilla algorithm however fits very well our architectural model, where we stream the values without needing to buffer them anywhere. The other benefit of it is that it is working algorithm for both floating points as well as integers. This simplifies the implementation of the compression job. Although as we can see from the value compression tests, some nature of the compression algorithm is not optimal to integer compression. For example, the Gorilla compression stores trailing zeros using 6 bits everytime the length of the leading zeros changes, but most of the integers do not have trailing zeros in the real world data, only leading zeros. That unfortunately causes those bits to be wasted.

Besides the technical merits of choosing the compression method, there was also a lot of positive buzz surrounding the Gorilla[92] compression algorithm when we started the project to implement a better compression method to Hawkular-Metrics. Multiple customers were referring to this and also we needed something that could be more easily understood by the users. By leveraging the idea that "Facebook uses this" the users quickly got the impression that this is good, instead of having to explain all the details on how the compression algorithm works. It is also used by multiple "competitors" which we talked about in the related work, which validated our approach to the users.

We did not want to limit ourselves to the Gorilla however and we made sure that all the parts can use a different compression method. Each datatype can have their own default compression algorithm and both timestamps as well as payload can be compressed with different algorithm. The data model allows to extend the compression metadata for future expansions also. We did not remove the LZ4 block compression from Cassandra, but instead decided to keep it. That allows us to compress the metadata and take care of certain patterns where Gorilla is not the best solution. One of these examples is the Openshift failed run, where we saw the relatively high overhead of Gorilla when it comes to the changing leading zeros or trailing zeros as that always costs 12 bits on top of the actual difference.

7.4 Architectural choices

The architectural choices that lead to the use of ephemeral (temporary) tables with unique names were not our first choice. Originally the idea of ephemeral tables used a solution of a ring buffer of tables, which allowed a very simplified method of calculating which table data is being written to and which one is being compressed. After the compression, the buffer's table would be dropped and recreated to avoid any tombstone issues mentioned earlier. The ring buffer approach had several advantages,

the lookups to find the correct table were faster as we could use an array of possible tables, the query calculations would always be at most circular and at most 12 tables. Also, there was no need to generate new prepared statements as the table names would be unchanged.

Although the initial implementation performed very well, there were two downsides. First, if the compression job for any reason would be more than 24 hours too late, the data would be written to a table that has already been filled and is waiting for compaction. This would mean more complex management in the compression job for this case as the job would have backbuffer the compression of this table and instead proceed on to the next one to catch up with the write load. The skiplist approach has no such issue, as it can work without any compression job by keeping a very large number of temporary tables lying around and they can be compressed at a later stage if required. This was not however the main reason of abandoning the ring buffer ideology, but the issues in the consistency of the Cassandra's schema changes[19] meant that we could not reliably use the same table name if there were nodes down while the operation of recreation took place. That would have caused schema uuid mismatches in the operation and corrupt the ring buffer management. These issues should have been fixed by the we rolled out the functionality, but that has not happened as of writing this paper.

Our first versions for customers also shipped with the use of a single data table instead of the temporary table storage mentioned here. We wanted to ship the compression to our customers as that was a heavily requested feature, but we did not have time to finish everything we wanted for those versions. We gained a lot of insight from those users and verified our assumptions for problems that might arise, especially when used without adequate hardware. For newer versions, we have replaced the solution with our temporary tables model.

7.5 Query performance

Increase in the query performance was not a reason for applying enhanced compression to the data and the only requirement was that it would not reduce performance by large margin. Thus, it was a positive surprise to see the reduction in query times. Although we see even better performance by moving to larger block sizes we did not at this point consider it very important for our block size selection. Looking at the Cassandra side tracing, we can see that there is no large difference in processing other than the amount of cells read, which are placed in a single sorted stream in the SSTable. There are issues in Cassandra handling larger partitions, so it is possible that upcoming versions of Cassandra could reduce the impact - we did test the newest snapshot version of 4.0 also, but it did not make a large difference.

There is an interesting outlier in the data processing if we would have used one hour blocks. Requesting less data is slower than requesting more data, which does not seem logical. We assumed this hiccup would go away with more tests, but more testing only confirmed the results. This is an interesting observation that might require some work in the Cassandra to remove this odd bottleneck. We did not notice huge difference in behavior when doing reads directly from disk with cleared

page caches, but this is most likely due to SSD not being the bottleneck in our tests. Using HDDs might render different results and enlarge the gains from reading less data.

The improvements in performance allowed us to remove a need for downsampling for older data. Many applications use downsampling for older data for faster queries by calculating an average for certain time range, but we can calculate this and other information from the raw values directly without a large performance penalty.

7.6 Future work

We do not expect to get large improvements to the compression ratio on our payload with optimization of the compression algorithms using the current approach. With the use of preprocessing and cascaded compression it is perhaps possible to gain 15-20% more, but this does not anymore transfer to large winnings. It would complicate the processing and comes with a performance cost. However, there are other parts of the system that could benefit from further improvements in the compression. The only exception to this is if we would allow users to define a lossy compression method, for example adjusting the precision based on the age of the data. In that case, old data could take a lot less space.

We did not touch the Cassandra's metadata compression at all in this paper. Instead, we keep relying on the LZ4 for the metadata compression, although we saw some 30% improvement when using Deflate in this case. However, the Deflate came with too large performance drop to our taste and as such we kept using LZ4. A more interesting approach could be to pregenerate dictionary for the Zstandard compressor to target the metadata, which does not change in our use-case as the table structure stays the same. That approach would benefit both the temporary table's structure as well as long term storage.

When it comes to the querying of the data, we currently uncompress it and then do the calculations to the data. For some use-cases this is not strictly necessary as we could compute a lot from the compressed data already. However, the more interesting use-case would be to share these compressed blocks directly to some third party application that could calculate the results. Such possibilities could involve large scale computing software such as Apache Spark[8] which could benefit from our data model to allow processing larger amounts of data in the memory. We have also interest in sharing our knowledge to applications that store traces such as traffic between distributed application components. Our team is investing in Jaegar[68], an opentracing compatible distributed tracing system that stores data in the Cassandra. It would require porting our processes to Golang, but the overall architecture should work there also to provide higher query speeds.

Performance wise, there are possibilities with Java 9 and Java 10 due to the introduction of better auto-vectorization and a potential native vectorized operations. These should allow to increase the compression performance with algorithms that are built to support vectorized operations. That means while we did not select PFOR in this occasion, this might become an interesting choice in the future.

The next challenge for time series storage models is most likely machine learning

and how it needs to use historical data. Streaming solutions and their window of knowledge can only solve a subset of all possibilities. At this point it is practically impossible to predict if our choices will scale to those requirements or if for example Cassandra's generic approach to data modeling will become a bottleneck. We do not envision the machine learning to take over traditional monitoring solutions in near future however, given that large parts of the processes are already automated in monitoring space and machine learning must provide something much better first to become interesting to business critical missions.

8 Summary

In this paper we have implemented a solution to Hawkular-Metrics that reduces the amount of stored data as well as shown that customized payload compression can be implemented successfully on top of Cassandra without storing state inside our application or modifying the Cassandra itself. We have also shown that our compression methods are both effective in reducing the size of the data on the disk as well as the improved performance when executing queries against the data. Other solutions on the market have used single node state to achieve similar objectives, but our solution is the only one which can work on a distributed environment. This gives it a distinct advantage when it comes to the scaling with larger data sets that do not fit a single machine.

We implemented our solution by using temporary tables in the Cassandra that are later compressed in a distributed job. The job distributes the workload by using Cassandra's native token ranges, lightweight transactions and also allows to implement concurrency inside the Hawkular-Metrics by splitting token ranges to smaller parts. We can keep the solution stateless as all our operations are idempotent and can be re-executed as many times as it is necessary until we can drop the temporary table. Our data model follows the limitations and advantages of time window compaction in the Cassandra.

The data is stored inside Cassandra in a columnar format where a single Cassandra row includes information for hours of data. This data is then compressed by our modified Gorilla based compression algorithm that provides more than double the performance in compression compared to the original solution. We also provided a Simple-8b solution which uses a faster algorithm for finding the optimal block size from a stream of integers and which can be used inside our solution also as we allow different encoding strategies.

We demonstrated that our changes reduced the query times by more than 90% compared to the naive approach in storing data to the Cassandra as a row model. Our compression algorithms also reduce the original data to fractions of the original representation with a suitable method for timestamps, enums, integers and floating points. We know the context of the data, which allows us to use a better model for compression than generic compression methods that operate on block of data such as Cassandra's own compression methods.

All the advantages have been contributed as open source with the compression methods being implemented as standalone libraries. This allows to use parts of our solution in other products also, thus benefiting larger community. Our improvements to the Gorilla compression algorithm can also be translated to other implementations without the need of rearchitecting the solution. The data compression benchmarks can be used as a starting point for any solution that stores similarly behaving data, including scientific data, not just monitoring data.

References

[1] *An Explanation of the DEFLATE Algorithm*. http://www.gzip.org/deflate.html. (Accessed on 01.10.2017).

[2] Michael P Andersen and David E Culler. "BTrDB: optimizing storage system design for timeseries processing". In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016.

[3] Vo Ngoc Anh and Alistair Moffat. "Index compression using 64-bit words". In: *Software: Practice and Experience* 40.2 (2010), pp. 131–147.

[4] Vo Ngoc Anh and Alistair Moffat. "Inverted index compression using word-aligned binary codes". In: *Information Retrieval* 8.1 (2005), pp. 151–166.

[5] *Apache Cassandra*. http://cassandra.apache.org/. (Accessed on 19.10.2017).

[6] *Apache License, Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0. (Accessed on 18.09.2017).

[7] *Apache Solr -*. http://lucene.apache.org/solr/. (Accessed on 22.10.2017).

[8] *Apache Spark - Lightning-Fast Cluster Computing*. https://spark.apache.org/. (Accessed on 04.10.2017).

[9] *Architecture in brief*. http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/architecture/archIntro.html. (Accessed on 29.08.2017).

[10] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. "Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. 2016, pp. 583–598. DOI: 10.1145/2882903.2915231. URL: http://db.cs.cmu.edu/papers/2016/arulraj-sigmod2016.pdf.

[11] *Beringei is a high performance, in-memory storage engine for time series data*. https://github.com/facebookincubator/beringei. (Accessed on 04.10.2017).

[12] *Bitcoin Historical Data | Kaggle*. https://www.kaggle.com/mczielinski/bitcoin-historical-data. (Accessed on 29.10.2017).

[13] Martin Burtscher. "An improved index function for (D) FCM predictors". In: *ACM SIGARCH Computer Architecture News* 30.3 (2002), pp. 19–24.

[14] Martin Burtscher and Paruj Ratanaworabhan. "gFPC: A self-tuning compression algorithm". In: *Data Compression Conference (DCC), 2010*. IEEE. 2010, pp. 396–405.

[15] Martin Burtscher and Paruj Ratanaworabhan. "High throughput compression of double-precision floating-point data". In: *Data Compression Conference, 2007. DCC'07*. IEEE. 2007, pp. 293–302.

[16] Martin Burtscher and Paruj Ratanaworabhan. "pFPC: A parallel compressor for floating-point data". In: *Data Compression Conference, 2009. DCC'09*. IEEE. 2009, pp. 43–52.

[17] Martin Burtscher et al. "Real-time synthesis of compression algorithms for scientific data". In: *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE. 2016, pp. 264–275.

[18] *Cassandra Time Series Data Modeling For Massive Scale*. http://thelastpickle.com/blog/2017/08/02/time-series-data-modeling-massive-scale.html. (Accessed on 29.08.2017).

[19] *[CASSANDRA-10699] Make schema alterations strongly consistent - ASF JIRA*. https://issues.apache.org/jira/browse/CASSANDRA-10699. (Accessed on 22.08.2017).

[20] *[CASSANDRA-9666] Provide an alternative to DTCS - ASF JIRA*. https://issues.apache.org/jira/browse/CASSANDRA-9666. (Accessed on 14.08.2017).

[21] *[CASSANDRA-9754] Make index info heap friendly for large CQL partitions - ASF JIRA*. https://issues.apache.org/jira/browse/CASSANDRA-9754. (Accessed on 29.08.2017).

[22] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. "A big data modeling methodology for Apache Cassandra". In: *Big Data (BigData Congress), 2015 IEEE International Congress on*. IEEE. 2015, pp. 238–245.

[23] Peter Pin-Shan Chen. "The entity-relationship model-toward a unified view of data". In: *ACM Transactions on Database Systems (TODS)* 1.1 (1976), pp. 9–36.

[24] *ConcurrentSkipListMap (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html. (Accessed on 19.10.2017).

[25] *Consistent hashing*. http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/architecture/archDataDistributeHashing.html. (Accessed on 30.08.2017).

[26] *criteo/biggraphite: Simple Scalable Time Series Database*. https://github.com/criteo/biggraphite. (Accessed on 01.11.2017).

[27] *Cyanite — Cyanite Hidden Coffin documentation*. http://cyanite.io/. (Accessed on 01.11.2017).

[28] *Data replication*. http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/architecture/archDataDistributeReplication.html. (Accessed on 30.08.2017).

[29] Jeffrey Dean. "Challenges in building large-scale information retrieval systems: invited talk". In: *Proceedings of the Second ACM International Conference on Web Search and Data Mining*. ACM. 2009, pp. 1–1.

[30] Giuseppe DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[31] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello. "Searching web data: An entity retrieval and high-performance indexing model". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 10 (2012), pp. 33–58.

[32] Luca Deri, Simone Mainardi, and Francesco Fusco. "tsdb: A compressed database for time series". In: *International Workshop on Traffic Monitoring and Analysis*. Springer. 2012, pp. 143–156.

[33] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951 (Informational). Internet Engineering Task Force, May 1996. URL: http://www.ietf.org/rfc/rfc1951.txt.

[34] *dgryski/go-tsz: Time series compression algorithm from Facebook's Gorilla paper*. https://github.com/dgryski/go-tsz. (Accessed on 19.10.2017).

[35] Sheng Di et al. "Efficient time series data classification and compression in distributed monitoring". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2007, pp. 389–400.

[36] *Differences to original Gorilla paper.. Issue 17*. 2017. URL: https://github.com/facebookincubator/beringei/issues/17 (visited on 02/17/2017).

[37] Jarek Duda. "Asymmetric numeral systems as close to capacity low state entropy coders". In: *CoRR* abs/1311.2540 (2013). URL: http://arxiv.org/abs/1311.2540.

[38] Frank Eichinger et al. "A time-series compression technique and its application to the smart grid". In: *The VLDB Journal* 24.2 (2015), pp. 193–218.

[39] *Enabling Cluster Metrics | Installation and Configuration | OpenShift Container Platform 3.6*. https://docs.openshift.com/container-platform/3.6/install_config/cluster_metrics.html. (Accessed on 30.08.2017).

[40] *Encoding | Protocol Buffers | Google Developers*. https://developers.google.com/protocol-buffers/docs/encoding. (Accessed on 22.10.2017).

[41] Vadim Engelson, Dag Fritzson, and Peter Fritzson. "Lossless compression of high-volume numerical data from simulations." In: *Data Compression Conference*. Citeseer. 2000, p. 574.

[42] Moein Falahatgar et al. "Universal Compression of Power-Law Distributions". In: *CoRR* abs/1504.08070 (2015). URL: http://arxiv.org/abs/1504.08070.

[43] Rosa Filgueira et al. "Adaptive-CoMPI: Enhancing MPI-based applications' performance and scalability by using adaptive compression". In: *The International Journal of High Performance Computing Applications* 25.1 (2011), pp. 93–114.

[44] Freddy and Gabbay. *Speculative execution based on value prediction*. Technion-IIT, Department of Electrical Engineering, 1996.

[45] Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. "Differential FCM: Increasing value prediction accuracy by improving table usage efficiency". In: *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE. 2001, pp. 207–216.

[46] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. "Compressing relations and indexes". In: *Data Engineering, 1998. Proceedings., 14th International Conference on*. IEEE. 1998, pp. 370–379.

[47] Leonardo A Bautista Gomez and Franck Cappello. "Improving floating point compression through binary masks". In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 326–331.

[48] *Graphite*. https://graphiteapp.org/. (Accessed on 21.10.2017).

[49] *HashMap (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. (Accessed on 19.10.2017).

[50] *Hawkular*. 2017. URL: https://www.hawkular.org (visited on 04/17/2017).

[51] *Hawkular - Metrics Documentation*. http://www.hawkular.org/hawkular-metrics/docs/user-guide/. (Accessed on 30.08.2017).

[52] *hawkular/hawkular-agent: Hawkular Agents that can be used to monitor managed products*. https://github.com/hawkular/hawkular-agent. (Accessed on 22.10.2017).

[53] *HdrHistogram by giltene*. http://hdrhistogram.github.io/HdrHistogram/. (Accessed on 29.08.2017).

[54] *Heroic Documentation*. https://spotify.github.io/heroic/. (Accessed on 01.11.2017).

[55] *Home | Metrics*. http://metrics.dropwizard.io/3.2.3/. (Accessed on 19.10.2017).

[56] *How is data deleted?* http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/dml/dmlAboutDeletes.html. (Accessed on 15.01.2017).

[57] *How is data maintained? | Apache Cassandra 3.0*. https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataMaintain.html. (Accessed on 15.01.2017).

[58] *How is data written?* http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/dml/dmlHowDataWritten.html. (Accessed on 15.01.2017).

[59] *How We Scale VividCortex's Backend Systems - High Scalability -*. http://highscalability.com/blog/2015/3/30/how-we-scale-vividcortexs-backend-systems.html. (Accessed on 22.10.2017).

[60] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[61] Lawrence Ibarria et al. "Out-of-core compression and decompression of large n-dimensional scalar fields". In: *Computer Graphics Forum*. Vol. 22. 3. Wiley Online Library. 2003, pp. 343–348.

[62] *IBM Knowledge Center - Informix product overview*. https://www.ibm.com/support/knowledgecenter/en/SSGU8G_12.1.0/com.ibm.po.doc/po.htm. (Accessed on 22.10.2017).

[63] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. Aug. 1985, p. 20. ISBN: 1-55937-653-8. URL: http://ieeexplore.ieee.org/iel1/2355/1316/00030711.pdf;%20http://standards.ieee.org/reading/ieee/std/busarch/754-1985.pdf;%20http://standards.ieee.org/reading/ieee/std_public/description/busarch/754-1985_desc.html;%20http://www.iec.ch/cgi-bin/procgi.pl/www/iecwww.p?wwwlang=E&wwwprog=cat-det.p&wartnum=019113.

[64] *InfluxData | Documentation | Storage Engine*. https://docs.influxdata.com/influxdb/v1.3/concepts/storage_engine/. (Accessed on 19.10.2017).

[65] *InfluxData (InfluxDB) | Time Series Database Monitoring & Analytics*. https://www.influxdata.com/. (Accessed on 04.10.2017).

[66] *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. (Accessed on 05.11.2017).

[67] *Introduction To The Apache Cassandra 3.x Storage Engine*. http://thelastpickle.com/blog/2016/03/04/introductiont-to-the-apache-cassandra-3-storage-engine.html. (Accessed on 15.01.2017).

[68] *jaegertracing/jaeger: Jaeger, a Distributed Tracing System*. https://github.com/jaegertracing/jaeger. (Accessed on 22.10.2017).

[69] *jdk8/jdk8/hotspot: 87ee5ee27509 src/share/vm/classfile/vmSymbols.hpp*. http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/tip/src/share/vm/classfile/vmSymbols.hpp. (Accessed on 05.11.2017).

[70] *KairosDB*. https://kairosdb.github.io/. (Accessed on 24.08.2017).

[71] S Klimenko et al. "Data Compression study with the E2 Data". In: *LIGO-T010033-00-E Technical Report* (2001), pp. 1–14.

[72] *kutschkem/fpc-compression: (Java-)Implementation of the compression method of Burtscher and Ratanaworabhan, "High Throughput Compression of Double-Precision Floating-Point Data"*. https://github.com/kutschkem/fpc-compression. (Accessed on 04.11.2017).

[73] *kutschkem/fpc-compression: (Java-)Implementation of the compression method of Burtscher and Ratanaworabhan, "High Throughput Compression of Double-Precision Floating-Point Data"*. https://github.com/kutschkem/fpc-compression. (Accessed on 15.10.2017).

[74] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[75] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*. Vol. 35. 5. ACM, 2000.

[76] Florian Lautenschlager et al. "Fast and efficient operational time series storage: The missing link in dynamic software analysis". In: *Proceedings of the Symposium on Software Performance (SSP 2015). Edited by G. eV. Softwaretechnik-Trends*. Vol. 3. 2015, p. 46.

[77] Daniel Lemire and Leonid Boytsov. "Decoding billions of integers per second through vectorization". In: *Software: Practice and Experience* 45.1 (2015), pp. 1–29.

[78] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. "SIMD compression and the intersection of sorted integers". In: *Software: Practice and Experience* 46.6 (2016), pp. 723–749.

[79] Peter Lindstrom and Martin Isenburg. "Fast and efficient compression of floating-point data". In: *IEEE transactions on visualization and computer graphics* 12.5 (2006), pp. 1245–1250.

[80] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. "Value locality and load value prediction". In: *ACM SIGPLAN Notices* 31.9 (1996), pp. 138–147.

[81] *LZ4 Block Format Description*. http://lz4.github.io/lz4/lz4_Block_format.html. (Accessed on 22.10.2017).

[82] *ManageIQ - Get Started*. http://manageiq.org/docs/get-started/. (Accessed on 19.10.2017).

[83] *Metrics 2.0: an emerging set of standards around metrics*. http://metrics20.org/. (Accessed on 19.10.2017).

[84] *Nagios - The Industry Standard In IT Infrastructure Monitoring*. https://www.nagios.org/. (Accessed on 22.10.2017).

[85] *Nanotrusting the Nanotime*. https://shipilev.net/blog/2014/nanotrusting-nanotime/. (Accessed on 30.08.2017).

[86] *Netflix/dynomite: A generic dynamo implementation for different k-v storage engines*. https://github.com/Netflix/dynomite. (Accessed on 22.10.2017).

[87] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[88] *OpenJDK: Panama*. http://openjdk.java.net/projects/panama/. (Accessed on 05.11.2017).

[89] *OpenNMS/newts: New-fangled Timeseries Data Store*. https://github.com/OpenNMS/newts. (Accessed on 01.11.2017).

[90] *OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes*. https://www.openshift.com/. (Accessed on 19.10.2017).

[91] *OpenTSDB - A Distributed, Scalable Monitoring System*. http://opentsdb.net/. (Accessed on 19.10.2017).

[92] Tuomas Pelkonen et al. "Gorilla: A fast, scalable, in-memory time series database". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1816–1827.

[93] *PKZIP | Data Compression | PKWARE*. https://www.pkware.com/pkzip. (Accessed on 01.10.2017).

[94] *Primitive Data Types (The Java™ Tutorials > Learning the Java Language > Language Basics)*. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. (Accessed on 05.11.2017).

[95] *Product - ScyllaDB*. http://www.scylladb.com/product/. (Accessed on 19.10.2017).

[96] *Prometheus - Monitoring system & time series database*. https://prometheus.io/. (Accessed on 04.10.2017).

[97] Piotr Przymus. "Query optimization in heterogeneous CPU/GPU environment for time series databases". In: (2014).

[98] William Pugh. *Concurrent maintenance of skip lists*. University of Maryland, Inst. for Advanced Computer Studies, 1990.

[99] *rackerlabs/blueflood: A distributed system designed to ingest and process time series data*. https://github.com/rackerlabs/blueflood. (Accessed on 01.11.2017).

[100] Jinfeng Rao, Xing Niu, and Jimmy Lin. "Compressing and decoding term statistics time series". In: *European Conference on Information Retrieval*. Springer. 2016, pp. 675–681.

[101] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. "Fast lossless compression of scientific floating-point data". In: *Data Compression Conference, 2006. DCC 2006. Proceedings*. IEEE. 2006, pp. 133–142.

[102] *ReactiveX*. http://reactivex.io/. (Accessed on 30.08.2017).

[103] *ReactiveX/RxJava: RxJava - Reactive Extensions for the JVM - a library for composing asynchronous and event-based programs using observable sequences for the Java VM*. https://github.com/ReactiveX/RxJava. (Accessed on 30.08.2017).

[104] *RealTime Data Compression: LZ4 explained*. http://fastcompression.blogspot.fi/2011/05/lz4-explained.html. (Accessed on 22.10.2017).

[105] Galen Reeves et al. "Managing massive time series streams with multi-scale compressed trickles". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 97–108.

[106] Jorma Rissanen and Glen G Langdon. "Arithmetic coding". In: *IBM Journal of research and development* 23.2 (1979), pp. 149–162.

[107] *RRDtool - About RRDtool*. https://oss.oetiker.ch/rrdtool/. (Accessed on 22.10.2017).

[108] Yiannakis Sazeides and James E Smith. *Implementations of context based value predictors*. Tech. rep. Technical Report ECE-97-8, University of Wisconsin-Madison, 1997.

[109] Yiannakis Sazeides and James E Smith. "The predictability of data values". In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society. 1997, pp. 248–258.

[110] Eric R Schendel et al. "ISOBAR preconditioner for effective and high-throughput lossless data compression". In: *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE. 2012, pp. 138–149.

[111] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. "Fast integer compression using SIMD instructions". In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. ACM. 2010, pp. 34–40.

[112] Falk Scholer et al. "Compression of inverted indexes for fast query evaluation". In: *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2002, pp. 222–229.

[113] Ilari Shafer et al. "Specialized Storage for Big Numeric Time Series." In: *HotStorage* 13 (2013), pp. 1–5.

[114] Fabrizio Silvestri and Rossano Venturini. "VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming". In: *Proceedings of the 19th ACM international conference on Information and knowledge management*. ACM. 2010, pp. 1219–1228.

[115] *Smaller and faster data compression with Zstandard | Engineering Blog | Facebook Code*. https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/. (Accessed on 02.10.2017).

[116] *Start page – collectd – The system statistics collection daemon*. https://collectd.org/. (Accessed on 22.10.2017).

[117] Alexander A Stepanov et al. "SIMD-based decoding of posting lists". In: *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM. 2011, pp. 317–326.

[118] *Storage engine*. http://docs.datastax.com/en/archived/cassandra/3.x/cassandra/dml/dmlManageOndisk.html. (Accessed on 15.01.2017).

[119] *The column-store pioneer | MonetDB*. https://www.monetdb.org/Home. (Accessed on 05.11.2017).

[120] *The gzip home page*. http://www.gzip.org/. (Accessed on 19.10.2017).

[121] *The Java HotSpot Performance Engine Architecture*. http://www.oracle.com/technetwork/java/whitepaper-135217.html. (Accessed on 05.11.2017).

[122] *Timescale | About*. https://www.timescale.com/about. (Accessed on 01.11.2017).

[123] *Time-series compression (part 2) – Akumuli*. http://akumuli.org/akumuli/2017/02/05/compression_part2/. (Accessed on 19.10.2017).

[124] *TWCS part 1 - how does it work and when should you use it ?* http://thelastpickle.com/blog/2016/12/08/TWCS-part1.html. (Accessed on 15.01.2017).

[125] Daniel G Waddington and Changhui Lin. "A fast lightweight time-series store for iot data". In: *arXiv preprint arXiv:1605.01435* (2016).

[126] *What is a Container*. https://www.docker.com/what-container. (Accessed on 09.09.2017).

[127] *What is Kubernetes?* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. (Accessed on 09.09.2017).

[128] Hugh E Williams and Justin Zobel. "Compressing integers for fast file access". In: *The Computer Journal* 42.3 (1999), pp. 193–201.

[129] Ian H Witten, Radford M Neal, and John G Cleary. "Arithmetic coding for data compression". In: *Communications of the ACM* 30.6 (1987), pp. 520–540.

[130] Aaron D Wyner and Jacob Ziv. "The sliding-window Lempel-Ziv algorithm is asymptotically optimal". In: *Proceedings of the IEEE* 82.6 (1994), pp. 872–877.

[131] Hao Yan, Shuai Ding, and Torsten Suel. "Inverted index compression and query processing with optimized document ordering". In: *Proceedings of the 18th international conference on World wide web*. ACM. 2009, pp. 401–410.

[132] *Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution*. https://www.zabbix.com/. (Accessed on 22.10.2017).

[133] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

[134] Jacob Ziv and Abraham Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE transactions on Information Theory* 24.5 (1978), pp. 530–536.

[135] *Zstandard - Real-time data compression algorithm*. http://facebook.github.io/zstd/. (Accessed on 02.10.2017).

[136] Marcin Zukowski et al. "Super-scalar RAM-CPU cache compression". In: *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE. 2006, pp. 59–59.

A Optimized Java

When implementing a compression algorithm, the JVM's (Java Virtual Machine) abilities which usually help to optimize software can be a hindrance. Because of the nature of the JIT (Just In Time compiler), there are now three levels to monitor, one is the actual source code and its algorithms, then the bytecode that's produced for the JVM and finally the optimization algorithms provided by the JVM and how this all turns into machine native code. For example, the GC (Garbage Collection) pauses can cause performance hickups in this performance sensitive code and it might also cause cache pollution which further reduces the speed. For that reason, the amount of garbage generated by the compression job must be minimal. Other simplifications of Java, such as providing only signed integers [94] makes the implementation a slightly more complex than the equivalent C bit manipulation.

A.1 Intrinsics

Intrinsic methods are functions that are handled especially by the compiler, or in this case the JIT. Instead of using automatically generated instructions, it uses something like inlined function, but with even more intimate knowledge of the behavior. In the JVM, these are often machine code directly, where the CPU has a native available function to implement the feature.

To extract maximal amount of performance, use of intrinsics should be optimized. Intrinsics in the JVM are tied to method names in certain shared classes and the complete documentation of available intrinsic function is only available in the source code of the OpenJDK[69]. Each of those methods have two or more implementations, the default one is pure Java implementation and then in the HotSpot, there's intrinsic code available for certain types of hardware. For example, if we take a look at the implementation of Long.numberOfLeadingZeros in Figure A1:

```
public static int numberOfLeadingZeros(long i) {
    // HD, Figure 5-6
     if (i == 0)
        return 64;
    int n = 1;
    int x = (int)(i >>> 32);
    if (x == 0) { n += 32; x = (int)i; }
    if (x >>> 16 == 0) { n += 16; x <<= 16; }
    if (x >>> 24 == 0) { n +=  8; x <<=  8; }
    if (x >>> 28 == 0) { n +=  4; x <<=  4; }
    if (x >>> 30 == 0) { n +=  2; x <<=  2; }
    n -= x >>> 31;
    return n;
}
```

Figure A1: Java source code for Long.numberOfLeadingZeros

We notice that the method itself has no knowledge of any intrinsic options. Instead, when the JIT sees this method is invoked it checks if there is an intrinsic version available for the hardware that is being used to run this code and only invoke the Java version of the code if there is no optimized version available. In the case of numberOfLeadingZeros, what is actually called as seen in listing A2 in the x86-64 systems:

```
ins_encode %{
  __ lzcntl(\$dst\$\$Register, \$src\$\$Register);
%}
```

Figure A2: JVM source code for Long.numberOfLeadingZeros

Here we can see that the intrinsic version of the same call is actually a single instruction call to the CPU, that does the same operation in the hardware. LZCNT counts the number of leading most significant zero bits in the x86 architecture, introduced with the AVX extensions for Intel processors. If that instruction is not available, then BSR instruction is instead executed, which returns the position of highest set bit, which is then reduced from word size to get the same result. LZCNT is an extension to the BSR function and both can be executed at higher speed than the pure Java version. [66]

A.2 Inlining

HotSpot uses adaptive optimizations to solve the problem of compilation by exploiting the reality that most of the time in program is spent by executing a small amount of code. It analyzes the runtime to detect hot spots and focuses global native code optimizer on those parts. By avoiding to compile everything, the compilation time is reduced and the HotSpot has more information on how to optimize the execution. HotSpot not only compiles hot code to native code, but it also monitors the frequency of method invocations, which are common in Java. By doing extensive method inlining it allows the compiler to focus on optimizing larger amounts of code as well as reducing the amount of method invocations. [121] To verify that inlining process optimizes the hot code path, JIT logging was enabled to show the inlining events and using this information we verified that everything is inlined correctly.

A.3 Superword Level Parallelism

While undocumented, the JVM detects increasing amount of opportunities for Superword Level Parallelism (SLP) [75] or so called auto-vectorization. For auto-vectorization, this is currently only applied to the loops that can be unrolled, which in Java means counted loops, but there are also several intrinsics that use SIMD for faster operations, such as array filling or array mismatch checking. While Java 8 can only support vectorization of the main loop, Java 9 also allows vectorization of the

post loop. Future versions of Java will include Project Panama[88], which allows to code vectorized code directly using Java.

A.4 Generic

HotSpot provides a large range of smaller optimizations that are applied if code follows good guidelines. These include loop optimizations such as loop peeling, range check eliminations and loop unrolling. These can have great effects on the performance and they are usually applied if the code follows some basic guidelines, such as making array loop invariant and using constant stride for the index variable. When avoiding if-else clauses, switch clauses are not only easier to read often, but they can also be compiled to table lookups if the switch clause is small and uses indexes.

Special attention must always be paid to the amount of garbage created as GC can eat large amounts of CPU, while also halting the program execution. While this can be partially reduced by using off-heap objects, it is good to remember that using them can cause serialization / deserialization overhead and is not always a recommended approach.

B Source code repositories

All the software produced while creating this thesis has been published online.

They can be found in Github under two different repositories. Work that went directly to the Hawkular project can be found under the Hawkular-organization while more general purpose contributions can be found under my personal repository.

B.0.1 Repositories

- Hawkular-Metrics:

  https://github.com/hawkular/hawkular-metrics

- Gorilla compression library for Java:

  https://github.com/burmanm/gorilla-tsc

- 64-bit word length integers compression algorithms:

  https://github.com/burmanm/compression-int

- Rewrite of the FPC-algorithm in Java:

  https://github.com/burmanm/fpc-compression

- Python script to parse Performance monitor files and send them to Hawkular-Metrics:

  https://gist.github.com/burmanm/88caff94a64f8116d11bb3b14aac45b5

B.1 Licensing

All the contributions have been published under the Apache License 2[6]